Implementación de la política DynaPeerUnicast para sistemas VoD bajo el paradigma P2P

Universitat de Lleida
Escola Politècnica de Lleida

Autor: Javier Aguirre Arnal
Director: Dr. Fernando Cores

20 de enero de 2008

Para mi família, amigos y mi director Fernando, que me han apoyado durante el desarrollo del proyecto y de esta memória, a todos ellos, grácias.

En esa gran cuna de filosofos que es Grecia dicen que: "El que nada duda, nada sabe".

Prólogo

Nos encontramos en un momento de la história en el que muchos miembros de la sociedad disponen de conexiones a internet de banda ancha. Esta situación es propicia para la implantación de sistemas de Video-over-Demand (video bajo demanda).

Estos sistemas pretenden la difusión de contenidos multimedia a través de internet según los gustos de los usuarios. No obstante, a pesar del aumento en los anchos de banda de la red nos encontramos que aún resultan un tanto escasos en cuánto tenemos que resolver un problema de vídeo en tiempo real.

El coste de los sistemas LVoD (LargeVideo-on-Demand) supone una gran inversión, por esta razón proponemos una solución basada en tecnologías P2P, en concreto DynaPeer.

El objetivo de este proyecto es precisamente el buscar una solución con la que aprovechar los recursos de todos los usuarios presentes en la red para crear un servicio VoD estable, esto lo realizaremos incorporando a un prototipo de Servidor VoD centralizado nuestra implementación de DynaPeer.

El resto de la memória se distribuye en los siguientes capítulos.

En el primer capítulo "Introducción" nos serán introducidos términos como VoD (Video Over Demand). A continuación encontraremos una explicación basada en el paradigma P2P del cual se mostraran sus principales aspectos tanto técnicos como desde una visión mas social, es decir, como ha afectado el paradigma P2P a la sociedad actual. Por último en este capítulo nos será introducido el concepto de DynaPeer objetivo de esta memoria.

El segundo capítulo esta centrado en analizar el diseño que hemos elegido para implementar la política DynaPeer. Encontraremos explicaciones sobre las funcionalidades que nos ofrece DynaPeer y sobre su diseño interno, exponiendo por ejemplo la función que nos aporta la utilización de roles para cada uno de los peers.

En el tercer capítulo se muestra la implementación de la política, con una descripción de sus partes mas significativas y con breves explicaciones sobre el código.

En el cuarto capítulo tratamos la validación de dicho prototipo con una serie de pruebas para comprobar si nuestro prototipo cumple con los objetivos marcados.

En el último capítulo exponemos nuestras conclusiones y dejamos varias líneas abiertas de investigación para mejorar ciertos aspectos de DynaPeer.

Índice

1. Introducción

Resumen

En este capítulo veremos una visión general del vídeo bajo demanda y de el ámbito en el cual esta desarrollado y de métodos para satisfacer tales peticiones como sería el P2P (Peer-to-Peer). Por último introduciremos el concepto DynaPeer, el cual se pretende exponer en esta memoria.

1.1. Introducción al proyecto

El desarrollo de este proyecto esta centrado en la gestión de un sistema de VoD. Este tipo de sistemas proporcionan al usuario la posibilidad de solicitar servicios multimedia en tiempo real. Este tipo de servicio requiere de una cantidad de recursos importantes tales como un ancho de banda considerable y una gran capacidad de almacenamiento para guardar los contenidos multimedia.

1.1.1. Marco proyecto

Debido al aumento en el ancho de banda al cual pueden acceder los usuarios desde hace unos años y a la bajada de precios en los dispositivos de almacenamiento podemos pensar que esos dos problemas estarían mas o menos solventados. El problema llega a la hora de gestionar todos esos recursos de la manera mas eficiente posible dado que los recursos requeridos son elevados para mantener una QoS[1] y en cualquier momento de saturación nuestra red podría afectar a la calidad de reproducción de los contenidos y con ello al servicio ofrecido. Para evitar toda esta serie de inconvenientes se establecen una serie de pautas de funcionamiento en las cuales se deciden varios aspectos como quien sirve que, a quien y durante que tiempo. Estos cuatro aspectos son de vital importancia para mantener una calidad de servicio en la reproducción de un contenido multimedia.

Por otro lado, tenemos que los sistemas de VoD requieren de una gran cantidad de recursos para poder realizar su función. Estos sistemas demandan grandes anchos de banda para la transmisión de la información a través de internet como una capacidad de almacenamiento importante para albergar los diferentes contenidos multimedia.

Para ofrecer estos servicios VoD a gran escala es necesario implementar arquitecturas LVoD (Large-scale VoD) compuestos por cientos sino miles de servidores, lo que implicaría un altísimo coste.

1.1.2. Objetivos

Con el objetivo de abaratar este tipo de sistemas se buscan alternativas como la propuesta en esta memoria, una propuesta basada en el paradigma P2P, capaz de ofrecer estos servicios en internet.

[1] Quality of Service

El paradigma P2P nos resulta de gran utilidad dado a sus características. Su gran capacidad colaborativa lo convierte en un candidato ideal para nuestros fines dado que nuestro modelo busca obtener una colaboración entre diferentes usuarios para poder mantener el servicio VoD.

Aún teniendo un sistema eficaz como sería el P2P no podemos olvidar que en internet existen gran variedad de usuarios con diversas características. Para empezar nos encontramos ante lineas de ADSL (Asymmetric Digital Subscriber Line), esto quiere decir que los usuarios no disponen del mismo ancho de banda de salida como de entrada, por lo general un usuario siempre tendrá un ancho de banda saliente inferior al de entrada. Por otro lado que el ancho de banda sea asimétrico no es el único problema, también nos encontramos con que los diferentes anchos de banda no son constantes, es decir, el ancho de banda del que dispone un usuario no es estable todo el tiempo sino que sufre tanto subidas como bajadas dependiendo de la situación de la red en ese momento. Otra causa de problemas podría deberse al hecho de los diferentes anchos de banda existentes. No podemos generalizar y pensar que todos los usuarios disponen de un mismo ancho de banda, sino que este es totalmente heterogéneo y depende tanto de zonas geográficas como de simplemente los contratos ofrecidos por los ISP's (Internet Service Providers).

Nuestro objetivo es la implementación de una política de servicio denominada DynaPeer la cual está integrada en una arquitectura P2P híbrida. Esta arquitectura P2P híbrida significa que dispondremos de un servidor central que mantendrá la información en espera y responderá a peticiones sobre esa información, nos encontraremos también frente a una serie de nodos responsables de hospedar la información y cuyas conexiones estarán gestionadas únicamente por ellos, manteniéndose el servidor central alejado en mayor parte de las tareas de gestión del sistema.

El objetivo de este proyecto es el diseño de unas políticas de gestión para un sistema de VoD con una estructura DynaPeerUnicast. Estas se componen de diferentes clases que marcan el comportamiento de cada uno de los elementos que forman la estructura DynaPeerUnicast.

Los diferentes componentes del sistema estarían enmarcados en tres grandes clases, servidor principal, servidor virtual y clientes peer-to-peer.

Para poder llevar a cabo esta tarea se deberán definir una serie de controles de admisión, políticas de servicio, políticas de planificación y demás funciones auxiliares capaces de gestionar los recursos de cada uno de los equipos integrantes de la red.

En los capítulos posteriores se expondrán las diferentes soluciones adoptadas para tratar cada uno de los problemas como podría ser el hecho de elegir que peers deben servir una determinada petición o que parte del vídeo debe enviar cada uno hacía el cliente.

1.2. VoD

1.2.1. Visión general del sistema Vídeo-bajo-Demanda

VoD son las siglas de Vídeo Over Demand (Vídeo Bajo Demanda) o dicho de otra manera televisión a la carta.

Este sistema permite a un usuario acceder a diferentes contenidos multimedia de una manera personalizada, esto significa que el usuario no depende de unos horarios puesto que puede acceder a cualquier contenido cuando el desee.

Las actuales condiciones tanto de ancho de banda como de acceso a el han hecho que la investigación en estos tipos de sistema haya aumentado de manera significativa en los últimos años. Hoy en día multitud de compañías ofrecen líneas de ADSL (Asymmetric Digital Subscriber Line) a un precio mas o menos competitivo con lo que una mayor parte de la sociedad podría tener acceso a una serie de contenidos multimedia (vídeo) a través del sistema VoD.

En ejemplo de este sistema sería el comercializado por la compañía Telefonica S.A el cual es denominado Imagenio [Imag]. En la figura [Figura 3] podemos ver un esquema del funcionamiento general de Imagenio. En las figuras [Figura 1], [Figura 2] podemos ver dos capturas de pantalla de los menus de Imagenio.

Figura 1: Pantalla de Imagenio (1).

Figura 2: Pantalla de Imagenio (2).

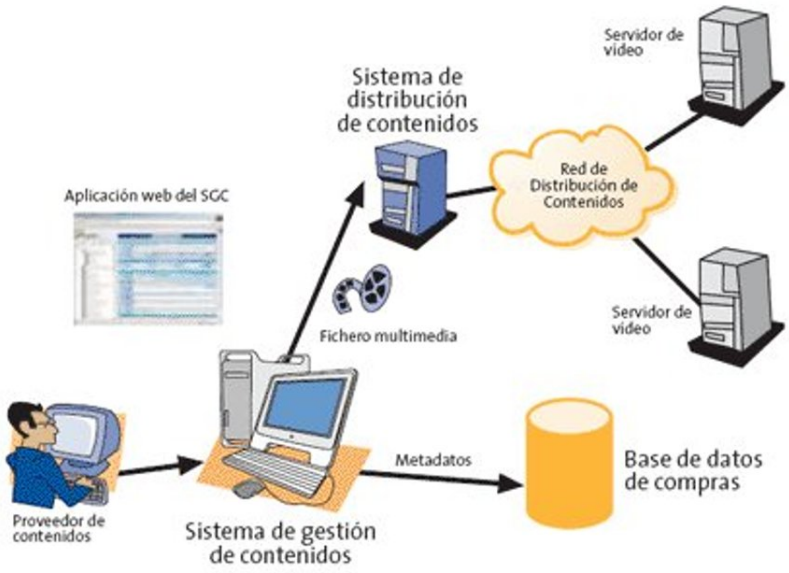

Figura 3: Funcionamiento general de Imagenio.

1.2.2. Componentes necesarios para un sistema VoD

Un sistema VoD esta apoyado en tres pilares, un servidor, una red por donde enviar los datos y una serie de usuarios que demandan información.

Servidor

La función del servidor sería la de almacenar los diferentes contenidos (vídeos) a los cuales un usuario podría tener acceso. El servidor debe proporcionar una serie de sistemas para que el usuario tenga una calidad en el servicio tras la petición de un contenido.

El servidor esta estructurado en tres partes (subsistemas): Subsistema de control, subsistema de almacenamiento y el subsistema de comunicación.

En la figura [Figura 4] se muestra un esquema que da idea de las diferentes partes que integran un servidor VoD.

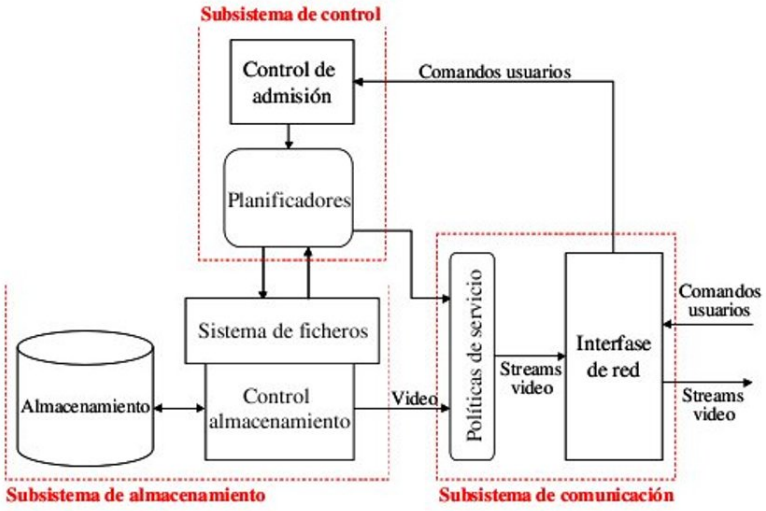

Figura 4: Modulos que componen un servidor de VoD [Cor04].

- Subsistema de control

  Este es el encargado de recibir las diferentes peticiones de usuario y mantener un orden de acciones para poder atenderlas.

  En esta parte es donde esta ubicada la política de control de admisión, la cual es la encargada de decidir si una petición puede ser servida o no. Este proceso en ningún caso debe afectar a las comunicaciones ya establecidas con anterioridad, dicho de otra manera, se debe mantener QoS (Quality Of Service) de las peticiones activas.

Otras de las funciones que le son atribuidas serían el control de las estadísticas del sistema y realización de procesos para la optimización de la eficiencia del sistema.

- Subsistema de almacenamiento

  Desde este módulo se guarda y se accede a los diferentes contenidos multimedia almacenados en los diferentes dispositivos de almacenamiento.

  La dificultad que entraña este módulo viene dada por el hecho que la información a tratar tiene un gran volumen, y esta información debe ser gestionada con una calidad de servicio (QoS) determinada por las diferentes aplicaciones de vídeo bajo demanda.

  El objetivo de este subsistema es el de aumentar la capacidad de ancho de banda del sistema de almacenamiento.

- Subsistema de comunicación

  Este módulo es el que se encarga de la planificación del envío de la información a través de Internet. En el están implementadas las diferentes políticas de servicio para poder optimizar los recursos de ancho de banda de la red ofreciendo de esta manera mayor volumen de información a un número mayor de usuarios.

Red de comunicación

La creciente mejora de la red de comunicación ha influido de una manera considerable en el aumento de aplicaciones multimedia.

Para que un servidor pueda servir las peticiones de los usuarios la red debe ofrecer una serie de condiciones tales como: mecanismos para el envío de las peticiones y demás mensajes necesarios para la comunicación y la posibilidad de poder enviar la información con una calidad de servicio (rendimiento).

En este aspecto VoD es un sistema realmente exigente con la red, puesto que requiere una gran cantidad de ancho de banda para transmitir volúmenes de información de gran tamaño a una velocidad elevada.

Clientes

Los requisitos fundamentales para un usuario serían que este soporte la recepción y la reproducción de los contenidos sin cortes, así como la opción de usar comandos VCR.

El interfase entre el sistema VoD y el usuario se denomina STB (Set-Top-Box), este es el encargado de recibir los comandos del usuario y enviar la señal al servidor a través de la red.

El STB almacena temporalmente los contenidos recibidos del servidor en unos buffers locales, decodifica la información en tiempo real y por último envía los contenidos a la pantalla de visualización.

Para realizar estas funciones el STB consta de 4 componentes esenciales: Interfase de red, decodificador, buffer y hardware de sincronización.

- Interfase de red

  Permite al cliente enviar información desde o hacía los servidores.

- Decodificador

  Para optimizar todo el sistema los datos multimedia suelen encontrarse codificados con lo que el cliente debe tener un medio para decodificarlos.

- Buffer

  La función de esta parte es de vital importancia para poder minimizar el daño producido por las condiciones de red. Esto se debe a que en la red de transmisión pueden producirse factores como congestión con lo cual resulta imposible saber en que momento llegaran los datos. Por esta razón se implementa un buffer en el que se guardara una cantidad de información que el cliente ira reproduciendo, recibiendo por tanto información mas adelantada a la que esta reproduciendo en el momento. Esta característica nos permite suavizar los requisitos de tiempo real en la transmisión y recepción del vídeo.

- Hardware de sincronización

  Los vídeos están compuestos por dos streams, uno de vídeo y otro de audio. Para la correcta reproducción ambos streams deben estar sincronizados entre si.

1.2.3. Requisitos de un sistema VoD

Un sistema de VoD entraña una complejidad técnica debido a que requiere que el servidor pueda manejar un tipo de contenido especial, en este caso multimedia, que pueda ofrecer una gestión de grandes volúmenes de información en tiempo real y mantener unos anchos de banda en la transferencia del disco tanto como de la red manteniendo con todo esto una calidad de servicio (QoS).

Los aspectos mas relevantes de cada uno de estos requisitos son:

- Gran capacidad de almacenamiento

  Los contenidos multimedia con los que nos encontramos en la actualidad debido a su calidad de imagen y audio son cada vez de un tamaño mayor llegando a requerir mas de una decena de gigas en algunos casos. Este hecho añadido a que un servidor debería gestionar una gran cantidad de contenidos hace que nos encontremos ante una situación en la que el orden de almacenamiento de un servidor sería del orden de TeraBytes.

- Servicio en tiempo real

  Garantizar la reproducción continuada de contenidos no consiste únicamente en que el servidor VoD envíe los datos y el cliente los reciba. Toda esta información debe realizarse en un tiempo determinado.

  Esto significa los componentes del sistema deben tener una noción temporal de los hechos. Para evitar problemas en los tiempos los diferentes componentes deben sincronizarse entre sí, no llevar a cabo esta sincronización convierte en imposible el poder ofrecer una calidad de servicio.

- Calidad de servicio (QoS)

La calidad servicio en un sistema de VoD consiste en tener una buena calidad de imagen y sonido, una buena recepción de ellos y una buena sincronización de ambos.

Todos estos aspectos no son fáciles de satisfacer puesto que para poder conseguir una QoS debemos obtener una armonía entre todos los componentes que integran el sistema VoD. Esto quiere decir que no solo basta con que cada uno de los componentes funcione correctamente de manera aislada, sino que una vez todos ellos actúan en conjunto ese rendimiento se debe mantener.

- Grandes anchos de banda

  Los contenidos multimedia con los que se trabaja en sistemas VoD requieren transmitir gran cantidad de información a través de la red de comunicación de una manera continuada con lo que es necesario tener acceso a una red con un elevado ancho de banda.

  Por otro lado a nivel mas interno del servidor los datos están guardados en dispositivos de almacenamiento a los cuales también se debe acceder para recuperarlos. Esto implica que dichos dispositivos deben ofrecer de igual manera que la red de transmisión una elevada capacidad de transmisión de datos.

  Obviar este aspecto en el diseño implicaría una sobrecarga del sistema VoD tras recibir un elevado número de peticiones.

1.3. Paradigma P2P

En este capítulo se realiza una introducción a los sistemas Peer-to-Peer (P2P).

Veremos que implicaciones tanto técnicas como sociales tienen este tipo de sistemas de compartición de información entre diferentes usuarios (peers).

1.3.1. Introducción

Dados los avances realizados en los últimos años tanto en incremento de velocidad en las líneas de Internet como en el acceso por parte de los usuarios a ellas se han conseguido desarrollar unos mecanismos de intercambio de información como podría ser el Peer-to-Peer que aprovechan este nuevo escenario de la sociedad.

La filosofía por la cual se rigen este tipo de conexiones es la de realizar un intercambio de información entre iguales, es decir, no existen ni servidores ni clientes sino una serie de nodos que se conectan entre ellos realizando funciones de servidor y de cliente concurrentemente.

Este tipo de redes son aplicables a situaciones tan diversas como la simple transferencia de archivos por parte de los usuarios de la red como labores de investigación en las cuales se podría realizar una computación distribuida debido a su estructura no centralizada.

Otra de las aplicaciones mas comunes para el P2P sería la denominada VoIP[2]la cual ha experimentado un creciente aumento en su uso.

1.3.2. Características

Las redes P2P presentan una serie de características que las diferencian de otro tipo de redes, estas características son precisamente las que le dan sentido, a continuación se describen.

- Escalabilidad

  El alcance de este tipo de redes es mundial dado que unos usuarios están conectados a otros formando una enorme red.

  Un factor básico para incrementar la eficacia de este tipo de redes es la cantidad de usuarios conectados a ella puesto que son estos últimos los que contribuyen con su recursos a aumentar los recursos disponibles de la red P2P.

- Robustez

  También se incrementa la robustez debido a su naturaleza distribuida en caso de la réplica excesiva de datos hacía múltiples destinos, y permitiendo a los peers encontrar la información sin hacer peticiones a ningún servidor centralizado de indexado.

- Descentralización

  Este tipo de redes son descentralizadas es decir, formadas por nodos iguales. Esta característica implica que ninguno de los nodos es imprescindible dado que no se le atribuyen características especiales a ninguno.

[2]Voice over Internet Protocol

- Costes repartidos

  Los recursos son donados y requeridos por todos los usuarios. Estos recursos tanto pueden ser en forma de información como en ancho de banda basándose toda la estructura P2P en esta colaboración entre usuarios.

- Anonimato

  Estas redes ofrecen un anonimato a los usuarios para proteger quien y desde donde esta realizando una determinada colaboración. Esta característica entra en conflicto con los derechos de autor los cuales pueden ser violados de una manera anónima por parte de cualquier usuario. Para evitar esta situación asociaciones de autores están intentando implantar sistemas de restricción de propiedad intelectual como el DRM[3].

- Seguridad

  Esta característica podría ser denominada como el "talón de Aquiles" de las redes P2P. Esto es así debido a la dificultad que implica identificar aquellos nodos que tienen unas intenciones no colaborativas con el resto de usuarios sino unas intenciones de daño al resto de la red.

  Se estudian varias opciones para intentar paliar este tipo de situaciones como crear grupos de nodos seguros en los cuales se pudiese desarrollar una colaboración P2P de manera segura conociendo a sus nodos.

1.3.3. Clasificación

La clasificación de este tipo de redes viene determinada por su nivel de centralización.

- Redes P2P centralizadas

[3]Digital Restrictions Management

En este tipo todas las gestiones se realizan a través de un único servidor central que sirve de punto de enlace entre los diferentes usuarios. Esto implica que el nivel de escalabilidad se ve disminuido, aunque las transferencias no sean centralizadas, debido a los recursos limitados que posee un servidor. Por otra parte la privacidad también se ve afectada dado que el servidor debe mantener dinámicamente un control de los nodos presentes.

Sus principales características son:

- Hay un único servidor el cual es el encargado de ser el punto de enlace entre los diferentes nodos y distribuir el contenido a petición de los nodos.

- Todas las comunicaciones dependen del servidor.

- Redes P2P totalmente descentralizadas

No requieren de ningún equipo central que las gestione. En este caso quien permite enlazar una conexión es un nodo que es un usuario que actúa como tal.

Sus principales características son:

- Los nodos realizan funciones de cliente y de servidor.

- Inexistencia de un servidor central.

- Carencia de un enrutador central que sirva como nodo y administre direcciones.

- Redes P2P híbridas

En esta versión de las redes P2P el servidor central actuaría a modo de hub, administrando el ancho de banda, funciones de enrutamiento pero siempre sin conocer las características de los nodos ni mantener información sobre las diferentes conexiones.

Una clara ventaja de este sistema la encontramos en el hecho de que si el servidor que gestiona todo cae, las conexiones existentes entre los diferentes nodos posibilitarían continuar con las transferencias de datos de una manera normal.

Sus principales características son:

- Posee un servidor central que mantiene información en espera y responde a peticiones sobre esa información.

- Los nodos son los responsables de hospedar toda la información.

- Los terminales de enrutamiento son direcciones usadas por el servidor, administradas por un sistema de índices para obtener una dirección absoluta.

1.4. Sistema DynaPeer

En este capítulo veremos los fundamentos en los cuales esta basado DynaPeer. Se expondrá su estructura y sus componentes.

Por último mostraremos sus ventajas respecto a otros sistemas.

1.4.1. Introducción

DynaPeer es una política de servicio diseñada para Internet. No es una política basada en un esquema cliente-servidor, sino que combina una estructura basada en la colaboración con un servidor y colaboraciones P2P entre los diferentes clientes.

La función del servidor sería la de almacenar todos los contenidos multimedia sobre los cuales un cliente podría realizar peticiones.

Otra función importante del servidor es la de establecer cada una de las nuevas conexiones por parte de un cliente desvinculándose posteriormente en alto grado sobre las transferencias que realice ese cliente obteniendo una estructura descentralizada. Esto implica que el servidor es el encargado de realizar una gestión global del sistema y de garantizar una QoS [SCY07].

Para conseguir un correcto funcionamiento de DynaPeer deberemos implementar una estructura basada en una jerarquía que en este caso toma forma a partir de una serie de roles. Según el rol que se tenga en cada momento cada uno de los peer deberá tomar unas decisiones u otras, variando totalmente la funcionalidad de un peer ante la misma situación dependiendo del rol que tenga en ese momento. Estos siempre serán asignados por el servidor principal dependiendo de las necesidades de cada momento.

1.4.2. Modelo de colaboración

El principio básico de DynaPeer es que los clientes son los que ponen a disposición sus propios recursos para formar un stream completo o parcial de vídeo con el que servir a nuevos clientes. Esto implica que en DynaPeer un cliente es capaz de reproducir un stream de vídeo y a la vez colaborar en la creación de otro para servir una petición.

Se ha considerado que los recursos de los que disponen los peers, ancho de banda y capacidad de almacenamiento, están limitados. En el caso del ancho de banda se presupone que es asimétrico, es decir, que el ancho de banda de entrada y el de salida no son iguales.

El ancho de banda de entrada debe ser como mínimo el mismo que el bitrate del vídeo y el ancho de banda de salida de un peer siempre sera inferior al bitrate de un vídeo.

En el apartado del almacenaje de los contenidos multimedia entra en juego un aspecto muy importante como es el del copyright. Este implica que un cliente no puede almacenar un contenido en su totalidad debido a los derechos de autor. Debido a esto se establece un buffer de almacenamiento limitado, basado en una ventana de colaboración desde la cual se ira reproduciendo un vídeo por parte de un cliente y a la vez marcara en que situación se encuentra para servir peticiones.

Todas las colaboraciones en DynaPeer están gestionadas por un Servidor Virtual, de ahora en adelante VS. Los VS son grupos de peers que se forman con el objetivo de agrupar recursos de diferentes clientes ó peers para poder servir peticiones de nuevos clientes de forma cooperativa. Entre sus funciones están la gestión de los recursos de los peers, la gestión de las peticiones y de las transferencias de datos consiguiendo así una red mas descentralizada al descargar al servidor principal de parte de la carga de peticiones.

Los recursos disponibles por un VS resultarían de la suma de recursos disponibles de cada uno de los peers que lo integran, siendo así por ejemplo su ancho de banda de salida igual a la suma de los diferentes anchos de banda de salida de sus peers [Figura 5].

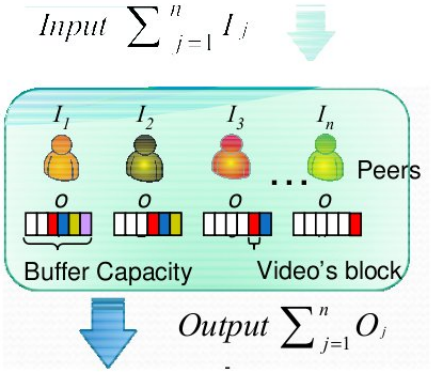

Figura 5: Recursos de un servidor virtual [Cor07].

El marco de colaboración de un VS viene definido por su ventana de colaboración la cual esta relacionada directamente con el buffer de almacenamiento del que disponen los peers integrantes del mismo.

Por tanto con estas premisas podemos definir un VS como una entidad definida por tres parámetros, *VS(j,s,w)* siendo s el offset del vídeo j durante un periodo de tiempo w.

Los VS siempre colaboran con el vídeo que están reproduciendo, es decir, los peers que lo integran reciben un stream de vídeo con el que van llenando sus respectivos buffers, estos peers van reproduciendo desde este buffer pero a su vez colaboran con la información que en el se guarda creando así la llamada ventana de colaboración la cual se va desplazando en función del tiempo.

1.4.3. DynaPeer Unicast

Dado que no todos los ISPs están provistos de tecnologías multicast en su acceso a internet, DynaPeer aprovecha para aplicar servicios unicast tanto en el lado del servidor como del el cliente.

En una configuración unicast, el servidor virtual está compuesto por un

número limitado de peers. Este número de clientes máximo viene determinado por la cantidad de clientes necesarios para servir una petición de forma completa, dato que resulta de la suma de anchos de banda de salida de los peers que integran el servidor virtual [Sou07].

DynaPeerUnicast define una forma de colaboración en la difusión de contenidos multimedia. Este tipo de estructura implica un diseño en cascada en el cual los contenidos se propagan a través de sus diferentes niveles.

En el caso de DynaPeerUnicast las conexiones que se realizaran entre los diferentes peers son de tipo unicast, es decir, se crea un canal entre servidor-cliente al que solo pueden acceder ellos y por el que se transmite la información.

Para mantener una conexión estable entre el cliente y el servidor no únicamente se crea un canal, sino que tenemos unos canales de control TCP a fin de asegurarnos que las informaciones importantes que entre ellos se deben comunicar lleguen sin problemas.

En la imagen [Figura 6] vemos el esquema general de un sistema DynaPeer unicast. Se muestra como inicialmente el servidor principal sirve las peticiones y como transcurrido un tiempo los primeros peers són capaces de servir una petición de play por parte del cuarto cliente.

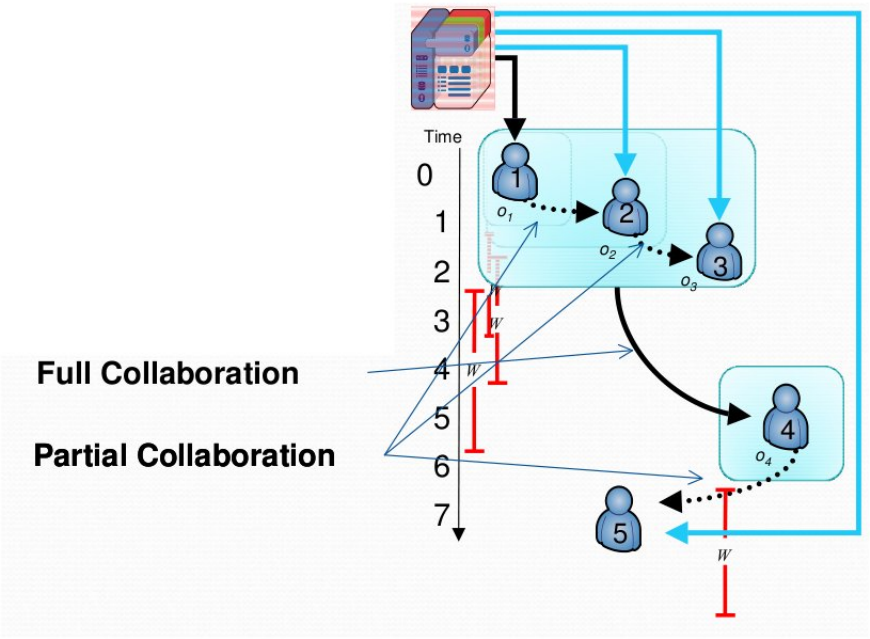

Figura 6: Esquema de DynaPeerUnicast [Cor07].

1.5. Prototipo servidor VoD

A fin de poder llevar a cabo nuestro proyecto disponemos de un prototipo del servidor principal desarrollado en la UAB por el grupo CAOS+Telefónica dentro de un proyecto profit [PRF05].

Este prototipo tiene una série de características que lo hacen idóneo para nuestro fin. Entre las mas importantes nos encontraríamos con un servidor multithread que nos permite múltiples conexiones de clientes pudiéndolas gestionar todas ellas de la forma mas eficiente posible.

El prototipo permite el control de cada uno de los clientes que a el se conectan a través de mecanismo de control como podrían ser canales de control de tipo TCP.

Su diseño esta basado en una serie de unidades cada una encargada de una parte del sistema teniendo entonces una entidad capaz de enviar contenidos a diferentes clientes.

Para poder obtener este resultado debemos tener un control sobre el sistema de almacenamiento del disco del servidor, un subsistema de control y una parte que controle la comunicación con los clientes.

El encargado de gestionar el control de disco es denominado VoDAlmacenamiento y es a el a quién se debe acudir en aquellas situaciones en las que debemos obtener información de los contenidos a servir.

El control de peticiones es realizado por el subsistema de control, este es el encargado de manejar todas aquellas solicitudes que recibe el cliente.

Por último el control de la red sería el último paso imprescindible para poder transmitir la información. Sin este paso toda la tarea anteriormente hecha no tendría sentido ya que los datos planificados y recuperados del disco no serían enviados a los diferentes clientes.

El funcionamiento de este servidor desde el momento que recibe una petición hasta que la sirve sería primero ver si se puede servir esa petición, ubicarla con una determinada prioridad, en cuanto llega el turno de servirla acceder al disco para recuperar la información multimedia y por último enviarla al cliente.

Esta sería una descripción básica del prototipo de servidor VoD utilizado en el desarrollo de este proyecto.

2. Análisis y Diseño

En este capítulo analizaremos dos de los aspectos mas importantes del proyecto, su análisis y el diseño que hemos adoptado para solucionar los problemas que se nos plantean.

2.1. Análisis funcional

Para hacer posible la creación de toda la estructura DynaPeerUnicast tenemos que disponer de unos clientes con unas características especiales, es decir, clientes capaces de servir contenidos y no únicamente de recibirlo.

Para conseguir estos peers se hizo una integración entre los sistemas del servidor y del cliente convencional obteniendo de esta forma una nueva entidad.

Para poder implementar DynaPeerUnicast sobre este diseño del clienteP2P se han tenido que realizar una serie de modificaciones sobre este a fin de permitir una funcionalidad completa para las nuevas políticas de servicio implementadas.

Un punto importante era dotar al clienteP2P de la posibilidad de enviar mensajes a diferentes servidores, ya que hasta este momento solo era capaz de gestionar un único servidor y no contemplaba la posibilidad de ser servidor por varios peers. Esta variación nos permite que en un futuro un cliente pueda comunicar a todos aquellos que le sirven que quiere realizar un determinado comando VCR y que las implicaciones que suponga este comando sean propagadas a todos aquellos usuarios de red que se ven afectados por esa petición.

Otro cambio realizado esta vez tanto en el servidor como en el clienteP2P es la inclusión de un nuevo parámetro dentro de las funciones de play. Este ha sido introducido con la finalidad de indicar al servidor la obligatoriedad de servir una petición o si por otro lado le damos libertad para redirigir la petición. Implícitamente este nuevo parámetro nos permite romper posibles bucles de peticiones dentro de la red que con los parámetros anteriores si se podían llegar a producir. Una vez expliquemos el funcionamiento de DynaPeerUnicast mas detalladamente se puede deducir con facilidad la importancia de este flag.

2.1.1. Política servicio servidor principal

Lo que da sentido y forma a DynaPeer en la parte del servidor principal sería el establecer una nueva política de servicio la cuál será la encargada de decidir ante cada una de las nuevas peticiones recibidas que decisiones tomar para resolverlas.

El servidor principal para poder tomar decisiones de acuerdo con las ideas de DynaPeer debe poder implementar unas estructuras de datos que mantengan información sobre los servidores virtuales presentes en la red, debe ser capaz a su vez de gestionar dicha información y transmitirla en un determinado momento a aquellos clientes que puedan requerirla.

Esta nueva política de servicio debe poder acceder a una serie de mensajes que usará para comunicarse con los diferentes clientes para comunicarles los resultados de cada una de las decisiones que toma el servidor en las diferentes situaciones.

2.1.2. Política servicio peer

La política de servicio de los peers se encargará de gestionar los recursos de una serie de peers que integraran un grupo.

Esta política deberá tomar decisiones referentes al conjunto de peer que forman parte de ese grupo gestionando así los recursos no únicamente del peer que toma la decisión, sino de todos aquellos que controla.

Para poder usar esta política de servicio deberemos mantener una determinada información sobre las características generales del grupo, entre estas la mas importante sería una lista de los peers que integran el grupo y de sus propiedades para poder luego tomar decisiones en función de estos datos.

Desde estas políticas deberemos poder acceder a una serie de funcionalidades que nos permitan elegir los peers que por ejemplo deberán servir una petición o decidir que parte tendrá que servir cada uno.

Por otra parte también tendremos que tener los mecanismos necesarios para realizar la gestión de los usuarios que en un momento determinado tenga que controlar un peer.

2.1.3. Roles

Los roles que pueden tomar los diferentes peers determinaran en cada momento que funcionalidades, propiedades e importancia tienen dentro de la estructura DynaPeer. Estos roles están asignados por el servidor en cuanto

se producen situaciones especiales dentro de la red como conexión de nuevos peers, caídas de estos y otras situaciones que se salgan del uso normal.

Los tipos de roles que nos encontramos son los siguientes:

- RolUnknownPeer: este tipo marcado como desconocido sera el que asignaremos a un peer en aquellas situaciones en que un peer no tenga una funcionalidad básica definida, a parte también nos servirá para casos en que el rol este definido por defecto.

- RolManagerPeer: este sería uno de los roles mas importantes. Es el encargado de la gestión de un VS. Cada VS es poseedor de un RolManagerPeer el cual es el encargado de la gestión del VS en tareas como el control de admisión, la ejecución de la política de planificación y la toma de decisiones dentro del VS. Este rol a diferencia del anterior es de obligada presencia en cada uno de los VS debido a que el servidor principal siempre que tenga que realizar alguna consulta o alguna acción sobre ese VS a quien le comunicara dichos cambios sera al RolManagerPeer que es el único capaz de realizar dichas acciones.

- RolHelper: la función de estos peers es la de proporcionar la ayuda necesaria para servir un determinado contenido multimedia en aquellos momentos en los que se carece de los suficientes peers como para atender una petición. Cuando el rol de un peer es asignado como RolHelper este peer en cualquier momento puede recibir por parte del servidor principal un canal en el cual se produzca un patching para que posteriormente este peer con la nueva información de la que posee sea capaz de colaborar por ejemplo en la propagación de aquellos contenidos multimedia con una menor demanda.

- RolBackupPeer: la función que desempeñan estos peers dentro de un VS es muy clara, como su nombre indica son los encargados de ayudar en un momento de bajada del ancho de banda de salida del VS con su propio ancho de banda para así poder proporcionar el servicio necesario al cliente del VS manteniendo de esta manera la QoS. Estos peers reciben el stream de vídeo de igual manera que el resto de peers del VS, la diferencia es que en este caso los RolBackupPeer's no reenvían ese stream sino que permanecen a la espera de que en un momento determinado alguno de los peers que si esta sirviendo disminuya su ancho de banda de salida o sufra una caída del VS.

- RolActivePeer: este rol será asignado a todos aquellos peers que sean capaces de enviar stream de vídeo hacía un cliente, es decir, todos aquellos peers que se encuentran en nuestra red recibiendo información de algún peer y que sean capaces de propagarla.

2.2. Diseño

2.2.1. Introducción

El diseño de DynaPeerUnicast esta basado en la implementación de tres entidades básicas, servidor principal, servidor virtual y manager del servidor virtual.

- Servidor principal: es el poseedor de todos los contenidos multimedia y la entidad a la cual se dirigen todas las peticiones iniciales de los usuarios sobre un determinado contenido. Tiene capacidad de tomar decisiones y de crear comunicaciones tanto unicast como multicast obteniendo así una gran versatilidad la cual es usada por la gran variedad de características que los usuarios pueden tener.
  El servidor principal es el encargado en cada momento de decidir que acción realizar dependiendo de las políticas de servicio que estemos usando o de los parámetros que le pasemos.
  Mantiene información tanto de los usuarios presentes en el sistema como de los contenidos que están reproduciendo, consiguiendo de esta manera una mejor eficiencia a la hora de tomar decisiones que afecten al conjunto. El contener toda esta información aunque en algunos casos sea básica implica que en momentos no favorables, por ejemplo caídas de peers, podremos acceder a unos datos mínimos a partir de los cuales volver a reconstruir la estructura.

- Servidor virtual: son entidades formadas por un conjunto de peers. Estos peers se encuentran todos dentro de una misma ventana de colaboración, es decir, todos ellos realizaron una petición de un contenido dentro de un mismo margen temporal.
  Estas entidades de la misma manera que reciben un stream de vídeo por parte del servidor principal o de otro VS también tienen la función de crear un stream de salida por el cual se servirán las peticiones de los clientes que lleguen a posterioridad.
  El número de peers integrantes en un VS viene determinado por la

cantidad necesaria de ancho de banda para servir peticiones de un determinado contenido, una vez completado un VS el siguiente peer en llegar sera asignado como creador del siguiente VS.

- Manager del servidor virtual: como antes hemos comentado esta entidad la realiza un peer al cual se le ha asignado el rol de RolManagerPeer. Tener activadas estas funcionalidades implica el mantener una serie de datos del servidor virtual en el que se encuentra tales como, peers que lo integran, anchos de banda de salida de cada uno de ellos, rol y estado de estos peers así como otras que se podrían añadir en cuanto fuesen requeridas.
  En caso de sustitución de manager en un servidor virtual el ultimo manager debe comunicar a poder ser los datos que posee al siguiente manager o bien al servidor virtual, implicando esta última opción una mayor facilidad de implementación. En caso de ser una caída repentina del manager de un servidor virtual y no poder realizar de esta manera la transferencia de información el servidor principal debe tener una información mínima con la que poder asignar otro manager para el servidor virtual y que este pueda recopilar de nuevo los datos a partir del servidor principal.

En la figura [Figura 7] podemos ver la estructura básica que se crearía. Tenemos que en el nivel superior nos encontraríamos con el servidor principal y que a partir de él se crearían los diferentes servidores virtuales los que se estarían conectados entre sí a través de un esquema de cascada. Los peers de un servidor virtual en condiciones normales se usarían para servir a los clientes del siguiente servidor virtual propagando de esta manera el vídeo. También se muestra como cada uno de los servidores virtuales tiene su propio manager.

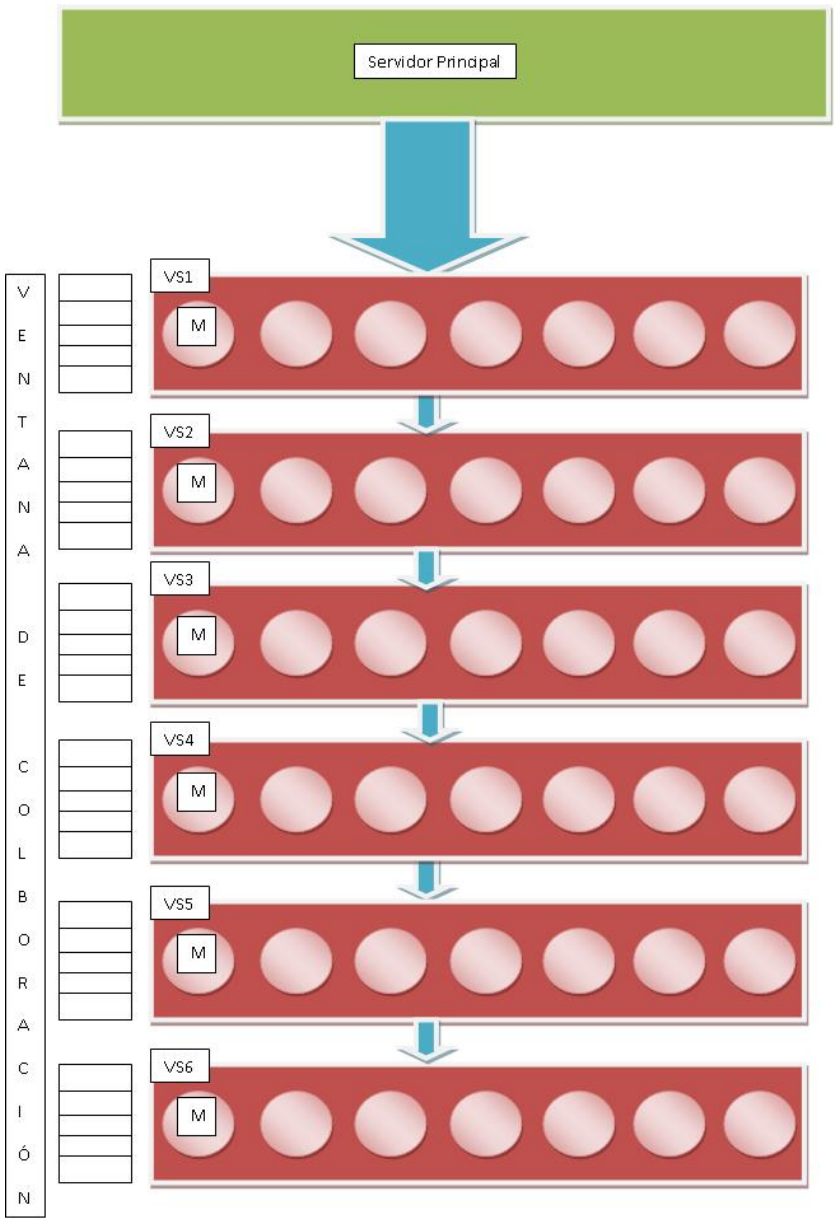

Figura 7: Esquema general de colaboración en DynaPeerUnicast

2.2.2. Información mantenida por cada entidad

- Servidor principal

- *Número de servidores virtuales:* Con este valor podremos acceder siempre de forma rápida a la cantidad de servidores virtuales controlados por el servidor principal.

- *Lista de servidores virtuales:* Esta lista nos permitirá mantener la información de cada uno de los servidor virtuales presentes en nuestra red.

- *Manager de cada uno de los VS:* Esta información estará contenida dentro de cada uno de los miembros de la lista de servidores virtuales. Esta será utilizada para saber a que peer de un VS debemos dirigir en cada momento nuestros mensajes de gestión del sistema.

- *Vídeo contenido por cada VS:* De la misma forma que el manager, cada elemento de la lista de servidores virtuales contendrá un valor que indicará el vídeo reproducido por ese servidor virtual.

- *Offset del vídeo contenido:* Esta información estará guardada de la misma forma que las dos últimas explicadas, es decir, en cada elemento de la lista de servidores virtuales. Indica que offeset del vídeo reproducido por un VS posee ese mismo servidor virtual.

- Servidor virtual

  La información contenida por el servidor virtual sería la que mantiene su manager puesto que un VS no es una entidad como tal, sino una unidad lógica dirigida por un peer que es el que guarda la información y toma las decisiones. La información referente al VS puede cambiar de usuario debido a posibles caídas o otras causas que pudiesen afectar al manager del servidor virtual.

- Manager del servidor virtual

  - *Número de peers que integran el VS:* Este valor nos indicará en cada momento la cantidad de peers que componen el servidor virtual.

  - *Lista de peers que integran el VS:* Con esta lista podremos en cada momento saber que peers forman parte de nuestro servidor virtual y las características de cada uno de ellos para poder tomar las decisiones correctas para la gestión del servidor virtual.

- *Ancho de banda de salida y entrada de cada uno de los peers del VS:* Esta información estará guardada dentro de cada uno de los elementos de la lista de peers. Para cada uno de los peers que integran la lista tendrémos sus anchos de banda pudiendo usar esta información para las gestiones del manager ante nuevas peticiones de play sobre el servidor virtual.

Por último cada uno de los peers al realizar un play mantiene información únicamente relacionada con el como serían los threads de control que tiene abiertos, consiguiendo de esta manera descentralizar del manager del VS una información que a priori no es necesaria que conozca.

2.2.3. Funcionalidades específicas de las entidades

- Servidor principal

  Entre las funcionalidades de un servidor principal se encuentra el control de almacenamiento, gestión de canales y demás gestiones de la red. A nivel de DynaPeer es el que primero recibe las peticiones y por tanto una pieza clave a la hora de ejecutar políticas de servicio puesto que una función muy importante que implementa es la de decidir quién y como va a servir una petición.

- Servidor virtual

  Las funcionalidades de esta entidad vienen marcadas por las del manager del servidor virtual dado que es este último el que da sentido y forma al VS, por tanto todo lo que pueda realizar un manager es lo que podremos pedir al VS.

- Manager del servidor virtual

  Entre sus funcionalidades se encuentran la capacidad de realizar un control de admisión sobre una petición de play, el poder ejecutar una política de planificación, tener capacidad de decisión a un nivel superior que el resto de peers integrantes de un VS

2.2.4. Resolución de peticiones de play

- Servidor principal

En el caso en el que el servidor principal reciba una petición de play por parte de un cliente, el servidor lo primero que hace es comprobar si es él el encargado de servir esa petición obligatoriamente o no. En caso de no estar obligado a servir esa petición la reacción del servidor es buscar un servidor virtual que se adecúe a la petición del cliente, es decir, un VS que este sirviendo un determinado vídeo en un offset parámetros los cuales son pasados por el cliente. Si el servidor encuentra un VS que pueda dar cobertura a esa petición simplemente le dice al cliente que redirija su petición de play hacia el VS asignado, en ese momento el cliente es quien debe repetir la acción de play pero en este caso contra al peer manager del VS que le ha indicado el servidor principal.

En el manager se evidencia una situación que diferencia la funcionalidad de un peer que simplemente realiza funciones de servidor respecto a la de un peer que realiza funciones de manager. En el caso de ser manager del servidor virtual cuando este recibe un play debe ejecutar un control de admisión en función del estado del VS posteriormente una selección de peers para servir la petición y por ultimo debe ejecutar la política de planificación, por otra parte en caso de ser un peer quién reciba el play únicamente realizara un control de admisión en función de sus propios recursos. De esta manera establecemos dos niveles de control, primero verificamos que el servidor virtual pueda servir la petición.

La siguiente imagen muestra cuál sería un proceso ideal en el que un cliente realiza una petición la cuál puede ser servida por un determinado servidor virtual. Los pasos están indicados en función del momento temporal en el que se ejecutan. A un nivel mas bajo se realiza el envío de una serie de mensajes que permiten llevar a cabo esta secuencia, estos mensajes no son mostrados en la ilustración [Figura 8] puesto que lo que se pretende es enseñar el proceso de una forma mas abstracta.

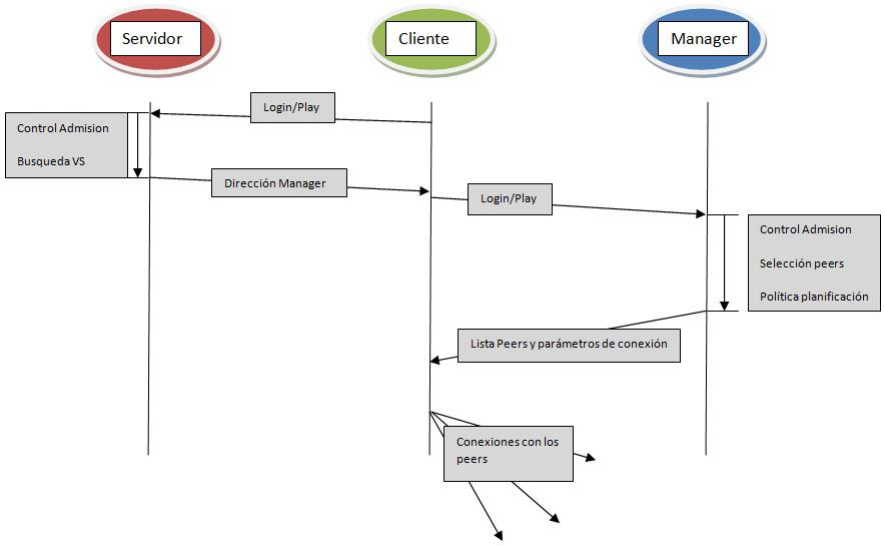

Figura 8: Esquema general del seguimiento de la resolución de un play.

- Servidor virtual

  Cuando un servidor virtual recibe una petición de play por parte de un cliente la primera tarea a realizar sería un control de admisión en el cual comprobaríamos si el VS dispone de los recursos necesarios para servir esa petición o si por el contrario se debe redirigir de nuevo hacía el servidor principal. El VS podrá servir esa petición siempre y cuando tenga la cantidad de peers necesarios como para crear un stream de un ancho de banda suficiente como para transferir el bitrate del vídeo.

  En caso de poder servir la petición el manager del servidor virtual envía al cliente una lista especificándole los peers a los que se debe conectar y que partes les debe pedir a cada uno de ellos.

  Una vez recibida esta lista el cliente realiza un play contra cada uno de los peers especificados por el manager.

  La siguiente imagen [Figura 9] muestra de forma mas detallada el último paso en la conexión del cliente con los diferentes peers indicado en la anterior ilustración [Figura 8].

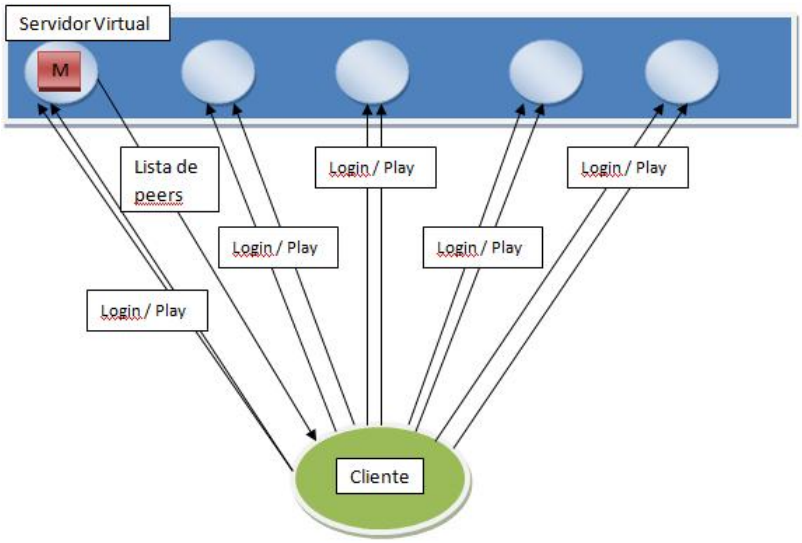

Figura 9: Esquema de play dentro de un VS.

- Manager del servidor virtual

  Este es el que recibe las peticiones iniciales redirigidas por el servidor principal, por tanto, la función del manager del servidor principal es la de realizar tareas de control de admisión y ejecutar una política de planificación una vez se ha aceptado una petición de play.

2.2.5. Resolución de situaciones de caída (acciones a tomar por cada entidad)

A continuación describiremos como se resolverían las situaciones de caídas de peers desde el punto de vista de cada una de las diferentes entidades.

- Servidor principal

  Si las caídas producidas no son de peers manager, el servidor principal únicamente se dedicará a eliminar estos peers de sus listas, es decir, actualizará sus tablas y borrará los canales que con ellos tuviese.

  En caso que la caída fuese de un manager el servidor debe identificar de que servidor era manager ese peer para poder ir a

su tabla de servidores virtuales inmediatamente a borrar ese peer de la lista y asignar un nuevo manager para ese VS. A la vez que se realiza esa nueva asignación el servidor también le debe transferir al nuevo manager la información básica a partir de la cual empezará a dirigir el VS.

- Servidor virtual

  Desde el punto de vista del servidor virtual cuando un peer que forma parte de el y sufre una caída el manager del servidor virtual se enterará de dicha caída a través del servidor principal el cuál si que mantiene conexiones con todos los peers presentes en la red.

  Una vez el manager recibe la notificación de caída de alguno de los peers que controlaba procede a eliminarlo de sus listas.

- Manager del servidor virtual

  Si se produce una caída de un peer que esta siendo cliente no habrá ningún problema para el VS que lo servía dado que cada uno de los peers que le estaban enviando información verá que sus correspondientes canales de control se han cortado y simplemente liberaran recursos y borraran dichos threads.

  En el caso que se produciese una caída por parte de un peer que esta sirviendo una petición a un cliente, el cliente será el primero en darse cuenta de la caída y deberá notificarla al manager del VS que le proporciona el stream. Ese manager será el encargado de notificar dicha caída al servidor principal el cual asignará en caso de poseerlos un backup peer que sustituirá al caído.

2.2.6. Diferentes casos de comandos VCR

- Stop

  Las situaciones en las que el cliente realiza un stop sobre un determinado vídeo se resolverían de forma similar a las de las caídas.

  En este caso al ser una situación controlada cada una de las entidades tiene un mayor tiempo de reacción para tomar las decisiones adecuadas a esta situación.

Cuando un peer recibe un stop por parte de un cliente lo gestiona de la misma forma que se gestionaría en una política unicast. La diferencia en DynaPeer viene cuando el servidor principal debe gestionar un stop.

Cuando el servidor principal recibe un stop por parte de un cliente antes de cortar todos los canales que le unen con ese cliente busca a que servidor virtual pertenecía ese cliente. A continuación borra a ese cliente de la lista de usuarios que tiene que forman parte del VS e informa al manager de dicho servidor virtual de que uno de sus peers ha sido eliminado.

En el caso de que el peer que realizase el stop fuese manager de algún VS la decisión que debe tomar el servidor principal una vez eliminado de la lista de usuarios del VS al que pertenecía, es la de asignar un nuevo manager de ese servidor virtual y enviarle los datos necesarios para que a partir de ese momento sea capaz en todo momento de gestionar los peers que forman parte de su virtual server. Con el envío de esta información conseguimos que la situación se estabilice de nuevo.

- Pause

  En situaciones de resolución de pauses el cliente continuará recibiendo información hasta llenar su buffer. Por tanto si un cliente que tiene un buffer de 10 minutos de vídeo podría realizar un pause de hasta 10 minutos sin tener efecto ninguno sobre el. Pasado este tiempo se cortarán los canales o el cliente deberá realizar de nuevo una petición de play a partir de un instante determinado puesto que en su buffer se habría tenido que borrar información todavía no reproducida por el usuario.

- Fast Rewind / Fast Forward

  La implementación de estas dos funcionalidades implica una gran complejidad debido a que una vez ejecutadas estas ordenes los peers deberían ir reubicándose en diferentes VS por momentos debido a que la ventana de colaboración del stream que reciben podría llegar a no abarcar lo suficiente en caso de pedir por ejemplo adelantar 10 minutos siendo la ventana de 5 minutos.

2.3. Política de selección de peers

Con el fin de que un servidor virtual pueda servir una petición a un cliente debe ejecutar una política de selección de peers. La finalidad de esta es la de escoger los mejores peers para servir una determinada petición.

La prioridad a la hora de escoger un peer u otro viene dada por el ancho de banda de salida que posee, así entonces al servir una petición siempre escogeremos los peers integrantes del servidor virtual cuyo ancho de banda sea mayor. Utilizaremos tantos peers como sean necesarios para servir la petición.

Para realizar esta función la lista de peers integrantes que mantiene el manager es ordenada de forma descendente según el ancho de banda de salida de cada uno de los peers, posteriormente seleccionamos los peers según ese orden hasta tener el ancho de banda necesario para cubrir la petición.

2.4. Política de planificación

Una vez realizada la selección de los peers que deben servir una petición debemos decidir que parte del vídeo debe servir cada uno de ellos. Esta decisión se toma en función de su ancho de banda de salida.

Para cada uno de los peers seleccionados calculamos el tanto por ciento que supone su ancho de banda de salida respecto al total necesario para servir la petición. Con este dato sabemos cuantos bloques debe servir de cada 100, el valor de 100 viene del módulo que aplicamos para conseguir que los peer sirvan stream de forma entrelazada. Los bloques son asignados a cada uno de los peers, enviando de esta manera al cliente una serie de mensajes de redirección de play hacía cada uno de los peers planificados con los parámetros que les debe solicitar del vídeo a cada uno de ellos.

El siguiente esquema [Figura 10] ilustra como se realiza la división del vídeo, en este caso el valor del modulo es 100, las diferentes celdas de la tabla representan la cantidad de bloques que tiene que servir cada peer en cada una de las secciones en que dividimos los bloques del vídeo, en el ejemplo el vídeo tendría un tamaño de 1000 bloques los cuáles tiene que ser servidos por cuatro peers.

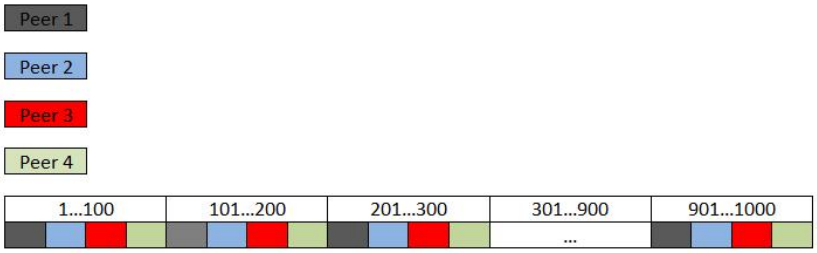

Figura 10: Planificación de un vídeo de 1000 bloques entre 4 peers

3. Implementación

En este capítulo veremos de que forma se ha implementado toda la estructura de DynaPeer anteriormente explicada. Sabremos de que forma se han resuelto los problemas anteriormente mencionados entre otras cosas.

3.1. Estructuras de cada una de las entidades

A continuación se hará una descripción de las estructuras de datos mantenidas por cada una de las diferentes entidades que forman parte del esquema DynaPeer.

3.1.1. Servidor

El servidor como base de todo el sistema debe mantener información de lo que en la red acontece, esto quiere decir que debe mantener información de cada uno de los servidores virtuales y algunos datos básicos de los usuarios, así como lógicamente información detallada sobre los contenidos multimedia.

3.1.2. Servidor virtual

El servidor virtual (es decir, su manager) guarda información acerca de los peers que lo integran. Los datos almacenados serían en número de peers integrantes del VS, sus identificadores y el ancho de banda de salida y entrada de cada uno de ellos.

3.1.3. Peers

Los peers que no tienen el rol de manager únicamente guardan información sobre cada uno de los threads de control que tienen abierto. Esta información es necesaria para que un peer pueda enviar peticiones simultaneas a cada uno de los peers que le están sirviendo.

La decisión sobre que información debe mantener cada uno de los peers esta basada únicamente en un criterio, su rol.

El rol nos indica que funcionalidades tendrá un peer, por tanto a su vez nos marca los datos necesarios a los que podrá acceder. Se ha optado por esta estrategia para poder disminuir el número de recursos usados por un

peer (aumento capacidad de almacenaje) y disminuir también el tráfico en la red al no tener tantos peers actualizando sus correspondientes datos.

3.1.4. Mensajes

Para la interconexión entre las diferentes entidades que integran un sistema DynaPeerUnicast se han tenido que definir una serie de nuevos mensajes que nos posibiliten la comunicación entre las partes.

La función de estos mensajes pasa por el intercambio de información entre servidor y peer o entre manager y peer.

- Mensajes Servidor-Peer

  - *Mensaje de asignación de rol:* El objetivo de este tipo de mensaje es informar al peer el rol que debe asumir en cada momento. Es un mensaje utilizado por ejemplo en el momento en el que un peer realiza una petición de play sobre el servidor. Si el servidor debe iniciar un nuevo VS para ese cliente le comunica mediante un mensaje de este tipo que su rol será el de manager para que este tomé las correspondientes acciones.

  - *Mensaje de redirección hacía VS:* Este mensaje está implementado con la finalidad de comunicar a un cliente donde debe realizar una petición de play, es decir, se da tras una petición de play del cliente al servidor en la cual el servidor encuentra un servidor virtual con las características apropiadas como para servir dicha petición. En la recepción de este mensaje el cliente vuelve a emitir una petición de play pero esta vez la realiza sobre el manager del VS indicado por el servidor.

  - *Mensaje de adición de un peer a un VS:* Este esta diseñado para ser utilizado después de la redirección de un peer hacía un determinado servidor virtual. Con esto conseguimos que el manager del servidor virtual tenga constancia de que un nuevo peer forma parte del servidor virtual que gestiona y así lo tenga en cuenta para futuras gestiones del VS.

- Mensajes Manager-Peer

- *Mensaje de redirección de play:* Este mensaje se envía una vez el servidor ha realizado la división del vídeo. Indica que porción debe solicitar el cliente a cada uno de los peers que le van a servir. Una vez el cliente recibe este mensaje lo trata de manera que compone diferentes peticiones de play hacía cada uno de los peers indicados.

3.1.5. Clases asociadas

Para poder facilitar la gestión de las diferentes entidades se han creado una serie de clases para facilitar la gestión de la información y así obtener a la vez una mayor flexibilidad para futuras ampliaciones.

Estas clases son utilizadas desde el servidor principal hasta los clientes pasando por los managers.

El servidor principal necesita mantener información detallada de los servidores virtuales presentes en la red, por otro lado un manager debe mantener información sobre los peers que integran el VS del cual es manager. Por último todos los peers tienen que tener una información básica para su própia gestión de recursos.

A continuación se detallan las diferentes clases y sus funcionalidades.

- *En el servidor:* El servidor mantiene información sobre los servidores virtuales presentes en la red. Para hacer posible esta funcionalidad se ha tenido que implementar en el servidor una clase que define los parámetros de un servidor virtual, creando posteriormente una lista de objetos de esta clase. El uso de esta clase posibilita una mejor gestión de todos los servidores virtuales en el sistema debido a que con ella podemos usar funciones típicas de las librerías stl[4] para trabajar sobre ella.

- *En el manager:* El manager debe mantener información para la gestión del servidor virtual. Para conseguir este objetivo se ha implementado una clase que define las características necesarias que debe conocer un manager de un usuario. Posteriormente hemos creado una clase que gestiona una lista de objetos de la clase

[4]Standard Template Library

anterior, ofreciendo a su vez nuevas funcionalidades que otorgar al manager.

4. Validación prototipo

A continuación se adjuntan una serie de capturas donde se muestra la evolución en la resolución de un play de dos clientes.

En la siguiente imagen [Figura 11] se ve como un cliente realiza una petición de play de un vídeo determinado. En este caso el servidor principal acaba de ser creado y no tiene ningún servidor virtual en sus tablas hacía el cuál redirigir al cliente, por tanto el servidor crea un nuevo servidor virtual con el cliente como manager (el servidor envía un mensaje al cliente diciéndole que es manager) y sirve la petición el directamente. En la ventana de la izquierda podemos ver enmarcado como el servidor principal crea un nuevo servidor virtual con los datos del cliente que realiza la petición de play y como posteriormente le asigna un nuevo rol al cliente. En la ventana del cliente (derecha) queda reflejado como el cliente recibe el mensaje de asignación de rol y cambia su rol en este caso al de manager.

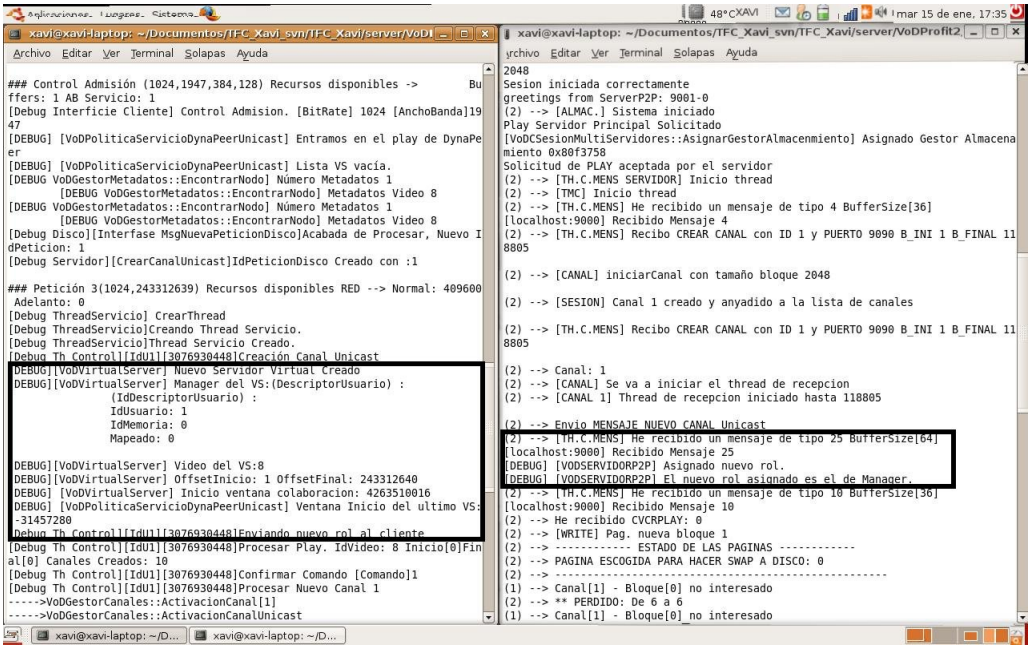

Figura 11: Cliente realiza captura al MainServer y este crea un nuevo VS.

A continuación podemos ver como el servidor principal tras crear el primer servidor virtual al recibir una petición de play idéntica a la del primer

cliente, es decir, el mismo vídeo inmediatamente después, busca un servidor virtual apropiado, en este caso si que lo encuentra y a continuación le envía la dirección de su manager al cliente [Figura 12].

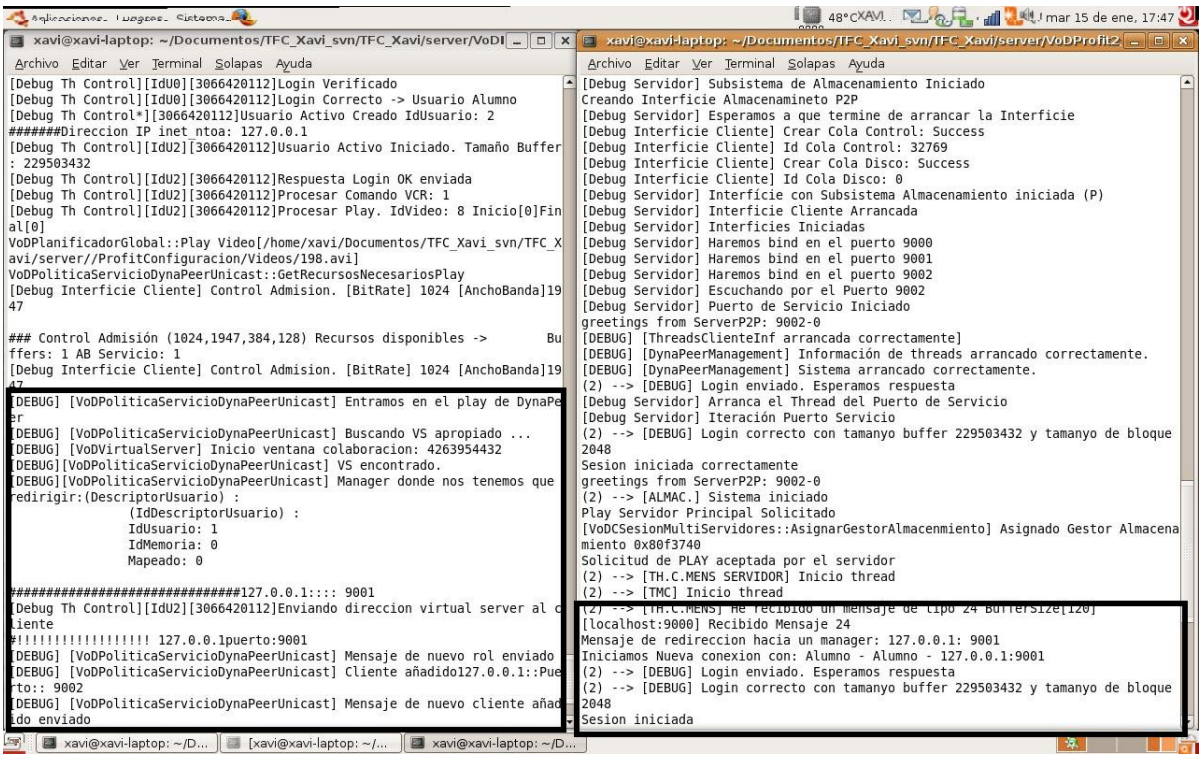

Figura 12: El MainServer busca un VS apropiado y redirige al cliente hacía el manager del VS.

A continuación el cliente toma la dirección del manager que le envía el servidor y realiza una nueva petición de play hacía el manager del servidor virtual. Para realizar esta nueva petición de play utiliza los mismo parámetros usados con el servidor. El manager corresponde a la ventana de la izquierda y el cliente a la de la derecha [Figura 13].

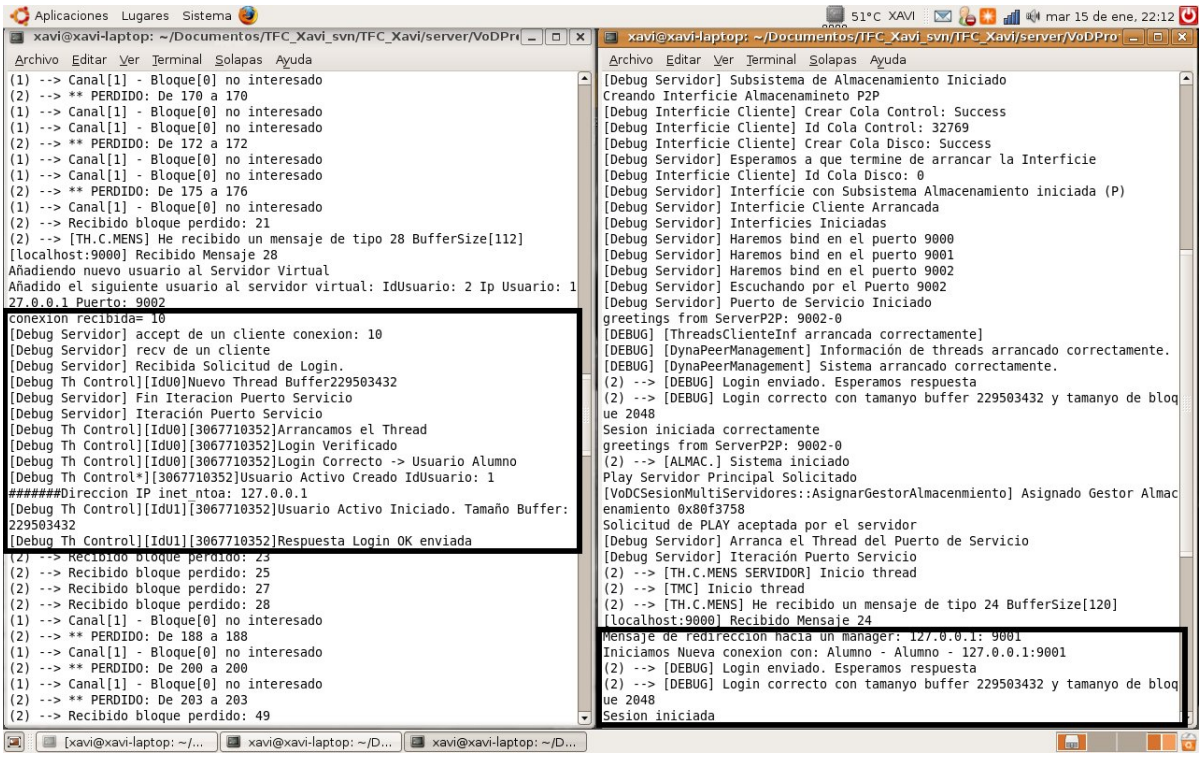

Figura 13: El cliente realiza una nueva petición de play al manager (manager a la izquierda).

Una vez el manager acepta el play del cliente realiza la ejecución tanto de la política de selección de peers que deben servir la petición como la política de planificación la cuál decidirá que parte del vídeo debe servir cada uno de los peers. En la ventana de la izquierda vemos enmarcados los mensajes resultantes una vez ejecutadas las políticas de selección de peers y de planificación [Figura 14].

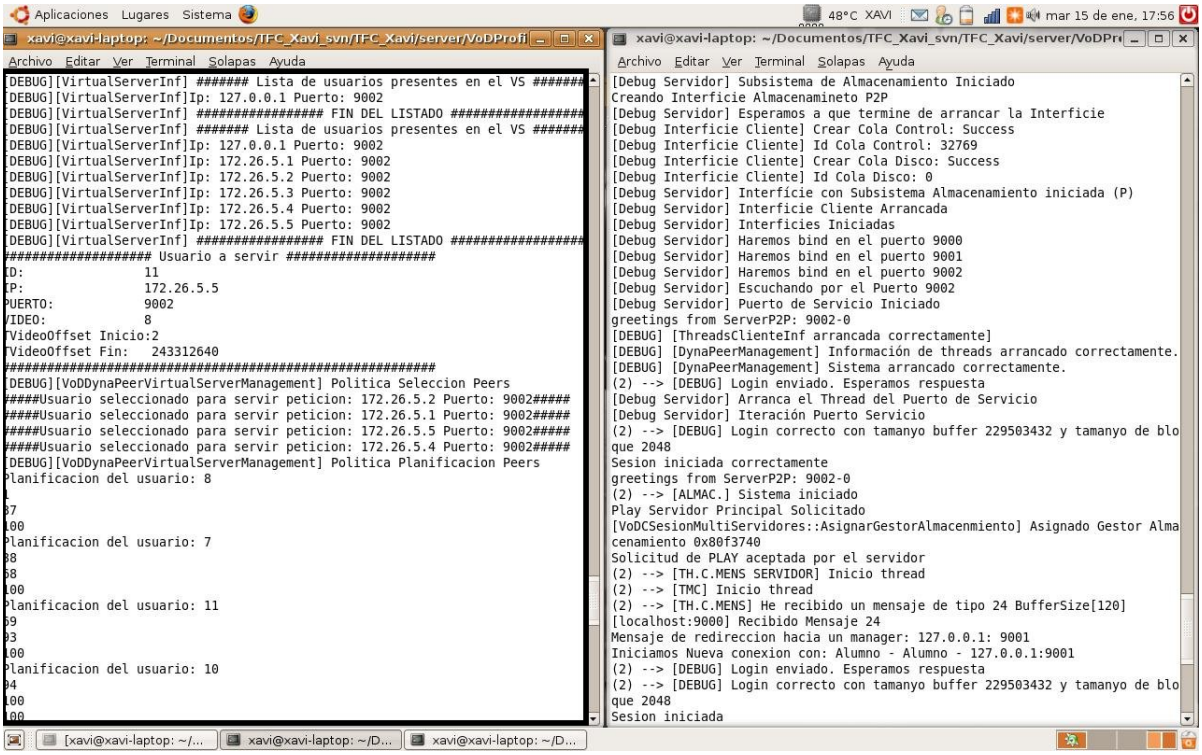

Figura 14: Ejecución de las políticas de selección de peers y planificación por parte del manager.

Seguidamente veremos el último paso en el proceso que es en el que el manager envía al cliente diferentes mensajes con los datos de los peers a los que se debe conectar y los bloques del vídeo que les debe pedir a cada uno de ellos, esto queda reflejado en la ventana izquierda correspondiente al manager. En la ventana del cliente (derecha) podemos ver como recibe un mensaje de redirección de play y realiza una nueva petición de play con los nuevos parámetros hacía el peer indicado por el manager [Figura 15].

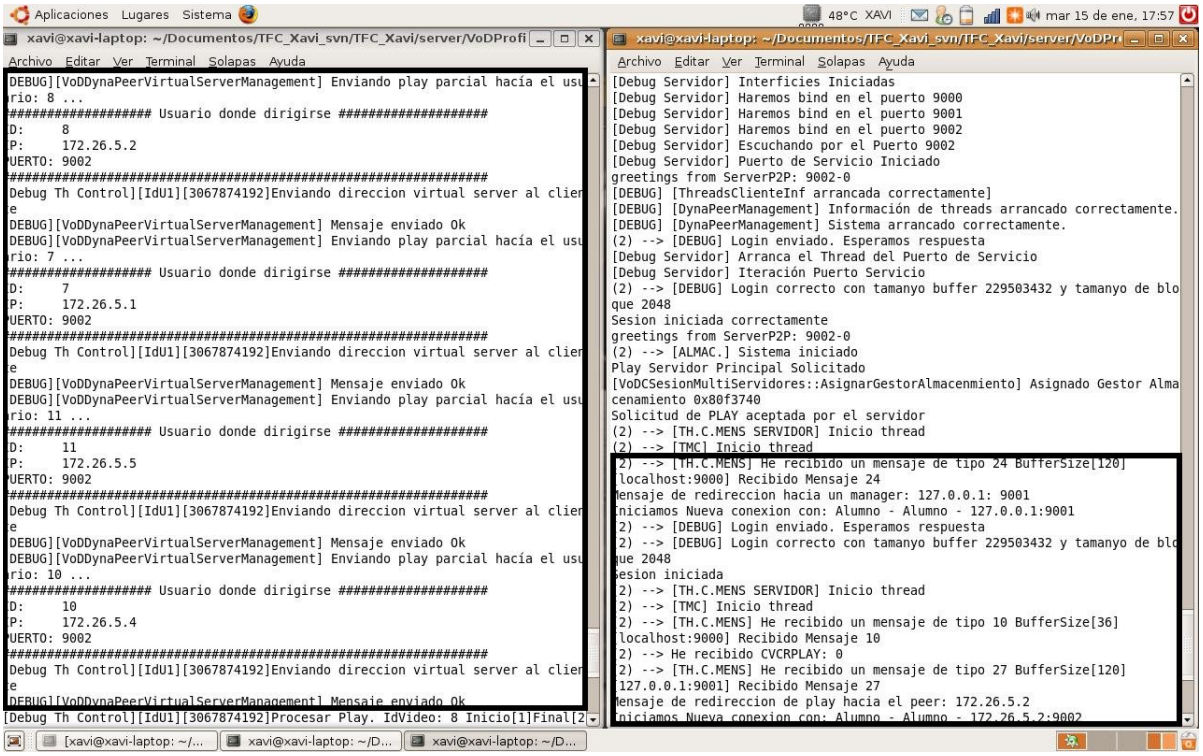

Figura 15: Redirección del cliente hacía múltiples peers para recibir el vídeo.

Para realizar el test tanto de la política de selección de peers como la de planificación se han añadido manualmente en la tabla del manager una serie de usuarios con unas determinadas características. Estos usuarios se pueden ver en la figura número 8.

5. Conclusiones y líneas abiertas

A medida que han ido pasando los meses desde que en una primera reunión con Fernando, mi director, me habló de una cosa que no entendí demasiado como era gestionar un sistema de vídeo bajo demanda a través del paradigma P2P, he ido tomando consciencia de lo amplia que puede llegar a ser esta área en cuánto a las opciones que podemos incluir.

Hoy en día se me pasan por la cabeza muchas ideas para poder mejorar el sistema como podrían ser el establecimiento de unos margenes en los anchos de banda de los peers que permitan dar una seguridad al sistema por ejemplo o diversas soluciones a situaciones de caída.

En cuanto a mi evolución personal y profesional durante este período considero un aumento considerable tanto en mi capacidad de análisis, diseño e implementación de soluciones. El uso de nuevas herramientas también ha sido un punto importante que tomar en consideración a la hora de hacer una valoración de lo que me ha aportado el proyecto.

El hecho de enfrentarme a un proyecto que se mantiene vivo desde hace un tiempo me ha obligado a tomar consciencia de lo importante que es el hacer las cosas bien pensando en la gente que vendrá después de nosotros y que deberá aprender de nuestro trabajo.

Como futuras ampliaciones del proyecto nos encontraríamos con que el paso natural sería desarrollar un sistema DynaPeerMulticast, añadiendo en este caso la implementación de helperPeers y backupPeers.

Otras posibles mejoras del sistema actual sería conseguir un sistema de tolerância a errores lo suficientemente firme como para poder gestionar aquellas situaciones que podrían degenerar en un desorden de toda la estructura mantenida por DynaPeer.

También sería interesante incluir nuevos parámetros de evaluación en el proceso de selección de peers por parte de un manager en el momento de servir una petición de play. La inclusión de valores que por ejemplo hiciesen referéncia a un histórico que nos indicase como es de estable el ancho de banda de un determinado peer nos ayudaría a elegir si queremos un peer con un ancho de banda elevado en un determinado momento o si por el contrario preferimos uno con un ancho de banda menor pero mas estable.

6. Referencias

[CAOS] http://caos.uab.es/

[Cor04] F. Cores: Arquitecturas Distribuidas para Sistemas de Video-bajo-Demanda a gran escala, Tesis Doctoral dirigida por la Dra. Ana Ripoll, UAB, 2004.

[Cor07] http://gcd.udl.cat/upload/recerca/PresNandoEUROPAR2007.pdf

[Imag] http://www.telefonica.es/sociedaddelainformacion/pdf/publicaciones/imagenio/capitulos/imageniocap9.pdf

[PRF05]PROFIT FIT-330301-2004-1: Plataforma de Servicios Multimedia para Centros Educativos. 2005-2007

[SCY07] L. Souza, F. Cores, XY. Yang, A. Ripoll: DynaPeer: A Dynamic Peer-to-Peer Based Delivery Scheme for VoD Systems. Euro-Par'07 Conference on Parallel Processing, LNCS, vol. 4641, pp. 769-781, 2007.

[Sou07] L. Souza: DynaPeer: A Dynamic Peer-to-Peer VoD System over Internet, Tesis Doctoral dirigida por la Dra. Ana Ripoll y el Dr. Fernando Cores, UAB, 2007.

Anexo A

En este apartado se adjunta todo el código nuevo que ha sido añadido al proyecto.

Anexo B - VoDPoliticaServicioDynaPeerUnicastServidor

Esta clase hace referencia a la política de servicio ejecutada en el servidor principal.

```
#ifndef __VoDPoliticaServicioDynaPeerUnicastServidor_H__
#define __VoDPoliticaServicioDynaPeerUnicastServidor_H__
#include <iostream>
#include <list>
#include <VoDBase.h>
#include <VoDTipos.h>
#include <VoDMetadatos.h>
#include <VoDServidor.h>
#include <VoDUsuarioActivo.h>
// #include <VoDPoliticaServicio.h>
#include <VoDPoliticaServicioUnicast.h>
#include <map.h>
#include <VoDVirtualServer.h>

/** Clase que implementa la política de servicio DynaPeer **/
class VoDPoliticaServicioDynaPeerUnicastServidor: public VoDPoliticaServicioUnicast{
        private:
         TListVoDVirtualServer VS;
         TListVoDVirtualServerIterator iter;
         int numOfVS;     //nos indica el numero de servidores virtuales creados.

    public:
         /* Constructora y destructora */

              VoDPoliticaServicioDynaPeerUnicastServidor();
        ~VoDPoliticaServicioDynaPeerUnicastServidor();

         /* Set y Gets */
         /* Función de inicialización de la política de servicio */
         virtual inline TCodigoError Inicializar()  { return(Error(CErrorNoImplementado)
              ); };

         /* Control Admisión */
         virtual TMbit GetRecursosNecesariosPlay(TDescriptorUsuarioActivo &usuario,
              TVideoId videoId, TVideoOffset inicio, TVideoOffset fin, TIdBloque
              BloqueRangoInicial, TIdBloque BloqueRangoFinal, TIdBloque ModuloRango);

         /* Comandos Reproducción Usuarios */
       virtual int Play(TDescriptorUsuarioActivo &usuario, TVideoId videoId,
            TVideoOffset inicio, TVideoOffset fin, TPosicionTurno ranuraTurno, TIdBloque
            BloqueRangoInicial, TIdBloque BloqueRangoFinal, TIdBloque ModuloRango, bool
            mandatory);

         virtual TCodigoError Stop(TDescriptorUsuarioActivo &usuario, TVideoId videoId);
       /*virtual TCodigoError BloquesPerdidos (TIdDescriptorUsuarioActivo &usuario,
            TVideoId videoId, TIdBloque IdBloquePerdido, TBloques NumeroBloquesPerdidos)
            */

    // Versión 2.0 Servidor
        inline virtual TCodigoError Pause(TDescriptorUsuarioActivo &usuario, TVideoId
             videoId){};
        inline virtual TCodigoError PauseActivo( TDescriptorUsuarioActivo & usuario,
             TVideoId videoId ){};
```

```
    inline virtual TCodigoError Goto(TDescriptorUsuarioActivo &usuario, TVideoId videoId
        , TVideoOffset desplazamiento) { Error(CErrorNoImplementado); return (
        CErrorNoImplementado); };

      // Soportado por el STB del usuario (No hay que implementarlo).
        inline virtual TCodigoError FastForward(TDescriptorUsuarioActivo &usuario,
            TVideoId videoId) { Error(CErrorNoImplementado); return (
            CErrorNoImplementado); };
        inline virtual TCodigoError FastRewind(TDescriptorUsuarioActivo &usuario,
            TVideoId videoId) { Error(CErrorNoImplementado); return (
            CErrorNoImplementado); };
       TDescriptorUsuarioActivo searchVS (TVideoId videoId, TVideoOffset off); //En caso
            de encontrar el VS adecuado retorna quien es su manager
    inline virtual void Monitorizacion(){for(iter=VS.begin(); iter!=VS.end();iter++)
             iter->imprimirInfoVS();} };

typedef class VoDPoliticaServicioDynaPeerUnicastServidor TPoliticaServicioDynaPeer, *
    PtrPoliticaServicioDynaPeer;
//Streams ostream &operator<<(ostream &stream, TPoliticaServicioDynaPeer &
    politicaServicio); ostream &operator<<(ostream &stream, PtrPoliticaServicioDynaPeer
    politicaServicio);
#endif          // __VoDPoliticaServicioDynaPeer_H__
```

Anexo C - VoDPoliticaServicioDynaPeerUnicastVS

Aquí se define la política de servicio a ejecutar dentro de un servidor virtual, es la utilizada por todos los peers.

```
#ifndef __VoDPoliticaServicioDynaPeerUnicastVS_H__
#define __VoDPoliticaServicioDynaPeerUnicastVS_H__
#include <iostream>
#include <list>
#include <VoDBase.h>
#include <VoDTipos.h>
#include <VoDMetadatos.h>
#include <VoDServidor.h>
#include <VoDUsuarioActivo.h>
#include "VoDPoliticaServicioUnicast.h"

/** Clase que implementa la política de servicio Unicast. **/
class VoDPoliticaServicioDynaPeerUnicastVS: public VoDPoliticaServicioUnicast {

  //TDescriptorMetadatos DescriptorMetadatos;
  //PtrMetadatos Metadatos;

   public:
 inline VoDPoliticaServicioDynaPeerUnicastVS(){ SetTipoObjeto(
     ObjVoDPoliticaServicioDynaPeerUnicastVS); SetPoliticaServicio(
     SrvPolDynaPeerUnicastVS);};                ~VoDPoliticaServicioDynaPeerUnicastVS()
     ;

      /* Set y Gets */
      /* Función de inicialización de la política de servicio */
     virtual inline TCodigoError Inicializar() { return(Error(CErrorNoImplementado)); };

      /* Control Admisión */
virtual TMbit GetRecursosNecesariosPlay(TDescriptorUsuarioActivo &usuario, TVideoId
    videoId, TVideoOffset inicio, TVideoOffset fin, TIdBloque BloqueRangoInicial=0,
    TIdBloque BloqueRangoFinal=0, TIdBloque ModuloRango=0);

      /* Comandos Reproducción Usuarios */
    virtual int Play(TDescriptorUsuarioActivo &usuario, TVideoId videoId, TVideoOffset
        inicio, TVideoOffset fin, TPosicionTurno ranuraTurno, TIdBloque
        BloqueRangoInicial=0, TIdBloque BloqueRangoFinal=0, TIdBloque ModuloRango=0,
        bool mandatory=false);

       virtual TCodigoError Stop(TDescriptorUsuarioActivo &usuario, TVideoId videoId);

     //virtual TCodigoError BloquesPerdidos (TIdDescriptorUsuarioActivo &usuario,
          TVideoId videoId, TIdBloque IdBloquePerdido, TBloques NumeroBloquesPerdidos);
```

```
    // Versión 2.0 Servidor
     virtual TCodigoError Pause(TDescriptorUsuarioActivo &usuario, TVideoId videoId);
     virtual TCodigoError PauseActivo( TDescriptorUsuarioActivo & usuario, TVideoId
         videoId );

   inline virtual TCodigoError Goto(TDescriptorUsuarioActivo &usuario, TVideoId videoId,
        TVideoOffset desplazamiento) { Error(CErrorNoImplementado); return (
       CErrorNoImplementado); };

   // Soportado por el STB del usuario (No hay que implementarlo).
   inline virtual TCodigoError FastForward(TDescriptorUsuarioActivo &usuario, TVideoId
       videoId) { Error(CErrorNoImplementado); return (CErrorNoImplementado); };
   inline virtual TCodigoError FastRewind(TDescriptorUsuarioActivo &usuario, TVideoId
       videoId) { Error(CErrorNoImplementado); return (CErrorNoImplementado); };};

typedef class VoDPoliticaServicioDynaPeerUnicastVS TVoDPoliticaServicioDynaPeerUnicastVS
    , *PtrPoliticaServicioDynaPeerUnicastVS;

//Streams ostream &operator<<(ostream &stream, TVoDPoliticaServicioDynaPeerUnicastVS &
    politicaServicio); ostream &operator<<(ostream &stream,
    PtrPoliticaServicioDynaPeerUnicastVS politicaServicio);

#endif          // __VoDPoliticaServicioDynaPeerUnicastVS_H__
```

Anexo D - VoDDynaPeerVirtualServerManagement

Esta clase implementa las funcionalidades de un manager y algunas básicas comunes al resto de peers.

```
#ifndef __VoDDynaPeerVirtualServerManagement_H__
#define __VoDDynaPeerVirtualServerManagement_H__
#include <iostream>
#include <list>
#include <VoDBase.h>
#include <VoDTipos.h>
//#include <VoDMetadatos.h>
//#include <VoDServidor.h>
//#include <VoDUsuarioActivo.h>
//#include <VoDPoliticaServicio.h>
//#include <stl_list.h>

//Estas dos nos serviran para mantener los direntes datos del servidor virtual
    dependiendo de si es manager o no.
#include <ThreadsClienteInf.h>
#include <VirtualServerInf.h>
// #include "VoDServidorP2P.h" #include <VoDSeleccionPeers.h>


/**  Clase que implementa las diferentes funciones de control de un VirtualServer. */
       //En esta clase se debe implementar una tabla con una visión mas específica del
           vs.     //Utilizar la tabla como base a todas las consultas, es decir
           utilizarla para comprobar si un usuario es el manager y si puede ejecutar
           determinadas acciones.

class VoDDynaPeerVirtualServerManagement: public VoDBase {
       private:

           //estas dos variables no deben ser listas, sino simples objetos de estas
               clases.          PtrThreadsClienteInf threads;
               PtrVirtualServerInf InfoServer;
           //En VoDServidorP2P esta implementada la vble que nos indica el rol de cada
               peer.

     public:
           VoDDynaPeerVirtualServerManagement();
          ~VoDDynaPeerVirtualServerManagement();
```

```
            int getSizeOfVS();

            bool ControlAdmision(TMbit RecursosNecesarios, TAnchoBanda BitRate);//control
                de admision global del vs

            void politicaPlanificacion(VirtualServerInf vsInf); //Se encarga directamente
                  de enviar al usuario los datos de los peers a los k se debe conectar y
                 que parte pedir a cada uno de ellos. Llamara a la función play parcial de
                  unicast.

            int DoStop(TDescriptorUsuarioActivo &usuario, TVideoId videoId);

      int DoPlay(TDescriptorUsuarioActivo &usuario, TVideoId videoId, TVideoOffset inicio
            , TVideoOffset fin, TPosicionTurno ranuraTurno, TIdBloque RangoModuloInicio,
           TIdBloque RangoModuloFinal, TIdBloque ModuloRango);

          inline void anadirUsuario(char Ip[50], TIdPuerto port, TPeerRol rol, bool
              supmul, TUserId iduser, TAnchoBandaServicio in, TAnchoBandaServicio out,
              TBloques buf){                    InfoServer->inputUser(Ip, port, rol,
              supmul, iduser, in, out, buf);};};

typedef class VoDDynaPeerVirtualServerManagement TVoDDynaPeerVirtualServerManagement, *
     PtrVoDDynaPeerVirtualServerManagement;
//Definir en VoDServidor la variable "static PtrVoDDynaPeerVirtualServerManagement
     VirtualManager" y la función "static inline void getVirtualManager(){return
     VirtualManager};" para poder acceder a las funciones de esta clase usando
     simplemente "VoDServidor::getVirtualManager()-> funcion"
#endif
```

Anexo E - Política Selección Peers

Esta es la clase que define como el manager debe escoger los peers para servir una determinada petición.

```
#ifndef __VoDSeleccionPeers_H__
#define __VoDSeleccionPeers_H__
#include<iostream>
#include<list>
#include<VoDBase.h>
#include<VoDTipos.h>
#include<VoDMetadatos.h>
#include<VoDVirtualServerUsers.h>

class VoDSeleccionPeers{
        private:
             TListVoDVirtualServerUsers selectedPeers;
             TAnchoBanda BWacumulado, BWnecesario;
        public:
           VoDSeleccionPeers(){BWacumulado=0;};
          ~VoDSeleccionPeers(){};

          inline TListVoDVirtualServerUsers seleccionarPeers (TUserId user,
               TListVoDVirtualServerUsers listaUsuarios, TAnchoBanda bw){
          //la lista de entrada nos vendra ordenada decrecientemente en funcion de BWout.
                                    TListVoDVirtualServerUsersIterator iter; //
               iterador necesario para recorrer la lista
            listaUsuarios.sort();
            BWnecesario=bw;

          for(iter=listaUsuarios.begin();iter!=listaUsuarios.end();iter++){ //Recorremos
               la lista de entrada
               if(BWacumulado<BWnecesario){ //miramos si el ancho de banda de salida
                    acumulado de los peers es suficiente
                         if(iter->getIdUsuario()!=user){

                              BWacumulado=BWacumulado+(iter->getBWout()); //Actualizamos el
                               ancho de banda acumulado
                                 selectedPeers.push_back(*iter); //Introduciomos el peer
                                      seleccionado en la lista de seleccionados
```

```
                                        cout<<"#####Usuario seleccionado para servir peticion
                                        : "<<iter->getIp()<<" Puerto: "<<iter->getPuerto()
                                        <<"#####"<<endl;

                                        }
                }
             else return selectedPeers;//cuando tenemos el ancho de banda necesario
                  retornamos la lista de peers seleccionados
             }

            return selectedPeers; //Este seria el caso en el que hemos recorrido toda la
                 lista y con todos los peers aun no tenemos suficiente ancho de banda para
                 servir la petición, de igual manera retornamos la lista puesto que todos
                 los peers estan seleccionados y el ancho de banda que falta lo aportará el
                  servidor.
  }

}; #endif //Fin de la classe VoDSeleccionPeers
```

Anexo F - Política Planificación

Clase encargada de realizar la división del vídeo entre los diferentes peers seleccionados por la clase anterior.

```
#ifndef __VoDPoliticaPlanificacionDynaPeer_H__
#define __VoDPoliticaPlanificacionDynaPeer_H__
#include <VoDPoliticaPlanificacion.h>
#include <VoDVirtualServerUsers.h>
#include "VoDPlanificados.h"
#include <list>

class VoDPoliticaPlanificacionDynaPeer: public VoDPoliticaPlanificacion {

     private:

  public:

   inline VoDPoliticaPlanificacionDynaPeer(){ SetTipoObjeto(
        ObjVoDPoliticaPlanificacionDynaPeer); SetPoliticaPlanificacion(PlPolDynaPeer); };
    ~ VoDPoliticaPlanificacionDynaPeer(){};

      TListVoDPlanificados planificacionVideo(TListVoDVirtualServerUsers selec,
           TAnchoBandaServicio bw);

       virtual inline TCodigoError Inicializar() { return(Error(CErrorNoImplementado));
            };
};

//Tipos typedef class VoDPoliticaPlanificacionDynaPeer TPoliticaPlanificacionDynaPeer, *
     PtrPoliticaPlanificacionDynaPeer;
//Streams ostream &operator <<(ostream &stream, TPoliticaPlanificacionDynaPeer &
     politicaPlanificacion); // osteram &operator <<(ostream &stream,
     PtrPoliticaPlanificacionDynaPeer politicaPlanificacion);

#endif  //__VoDPoliticaPlanificacionDynaPeer_h__
```

A continuación la función de planificación del vídeo.

```
#include <VoDPoliticaPlanificacionDynaPeer.h>
#include "VoDVirtualServerUsers.h"
#define modul 100

TListVoDPlanificados VoDPoliticaPlanificacionDynaPeer::planificacionVideo(
     TListVoDVirtualServerUsers selec, TAnchoBandaServicio bw){
        VoDPlanificados planificado;
        TListVoDVirtualServerUsersIterator iterator;    TListVoDPlanificados
             planificados;
        TAnchoBandaServicio anchoAcumulado=0, tantoxciento;
```

```
for(iterator=selec.begin();iterator!=selec.end()&&((*iterator).getIdUsuario()!=selec.
    back().getIdUsuario());iterator++){
        //Calculamos el % que corresponde a ese cliente
                tantoxciento=((iterator->getBWout())*100)/bw;
       //Lo sumamos a acumulado para saber en cada momento cuanto nos falta
                VoDPlanificados planificado((*iterator).getIdUsuario(),1+anchoAcumulado,
                    anchoAcumulado+tantoxciento,modul);

             cout<<"Planificacion del usuario: "<<(*iterator).getIdUsuario()<<endl;
             cout<<1+anchoAcumulado<<endl;
             cout<<anchoAcumulado+tantoxciento<<endl;
             cout<<modul<<endl;
//           planificado.setUser((*iterator).getIdUsuario());
//           planificado.setRangoModuloInicio(1+anchoAcumulado);
//           planificado.setRangoModuloFin(anchoAcumulado+tantoxciento);
//           planificado.setModuloRango(modul);
             planificados.push_back(planificado);
             anchoAcumulado=anchoAcumulado+tantoxciento;      }

            iterator++;
//           VoDPlanificados planificado((*iterator).getIdUsuario(),1+anchoAcumulado,
    modul,modul);

        planificado.setUser((*iterator).getIdUsuario());
  planificado.setRangoModuloInicio(1+anchoAcumulado);                planificado.
      setRangoModuloFin(modul);
        planificado.setModuloRango(modul);
        cout<<"Planificacion del usuario: "<<(*iterator).getIdUsuario()<<endl;
                                            cout<<1+anchoAcumulado<<endl;
        cout<<modul<<endl;

        planificados.push_back(planificado);
        return planificados;
}
```

Anexo G - VoDVirtualServerUsers

Esta clase define los datos que mantiene el manager de cada servidor virtual de cada uno de los peers que están dentro su VS. Los managers de los diferentes servidores virtuales implementan listas de objetos de esta clase.

```
#ifndef __VoDVirtualServerUsers_H__
#define __VoDVirtualServerUsers_H__
#include <VoDBase.h>
#include <VoDTipos.h>

/** Clase que define la información básica de un peer dentro de un ServidorVirtual. */

class VoDVirtualServerUsers{
private:
        bool                    supportMulticast;
        TPeerRol                Rol;
        TUserId                 IdUsuario;
        TAnchoBandaServicio     BWin;
        TAnchoBandaServicio     BWout;
        char                    ip[50];
        TIdPuerto               puerto;
        TBloques                buffer;

public:
   inline VoDVirtualServerUsers(){
             supportMulticast= false;
             Rol=RolUnknownPeer;
             IdUsuario= 0;
     }
```

```
        inline VoDVirtualServerUsers(char Ip[50], TIdPuerto port, bool multicast,
             TPeerRol rol, TUserId user, TAnchoBandaServicio bwin, TAnchoBandaServicio
             bwout){
    strcpy(ip, Ip);
    puerto=port;
    supportMulticast=multicast;
    Rol=rol;
    IdUsuario=user;
    BWin=bwin;
    BWout=bwout;
 }

        ~VoDVirtualServerUsers(){};

        //Sets & Gets
        inline void setsupportMulticast(bool supmul){ supportMulticast=supmul; }
        inline void setRol(TPeerRol rol){ Rol=rol; }
        inline void setIdUsuario(TUserId usuario){ IdUsuario=usuario; }
        inline void setBWin(TAnchoBandaServicio bwin){ BWin=bwin; }
        inline void setBWout(TAnchoBandaServicio bwout){ BWout=bwout; }
        inline void setIp(char Ip[]){ strcpy(ip, Ip); }
        inline void setPuerto(TIdPuerto port){ puerto=port; }
        inline void setBuffer(TBloques buf){ buffer=buf; }

        inline bool getsupportMulticast(){return supportMulticast;}
        inline TPeerRol getRol(){return Rol;}
        inline TUserId getIdUsuario(){return IdUsuario;}
        inline TAnchoBandaServicio getBWin(){return BWin;}
        inline  TAnchoBandaServicio getBWout(){return BWout;}
        inline char* getIp(){return ip;}
        inline TIdPuerto getPuerto(){return puerto;}
        inline TBloques getBuffer(){return buffer;}

inline VoDVirtualServerUsers const &operator=(VoDVirtualServerUsers const &usuario){
    supportMulticast=usuario.supportMulticast;
    Rol=usuario.Rol;
    IdUsuario=usuario.IdUsuario;
    BWin=usuario.BWin;
    BWout=usuario.BWout;
}

//Redefinimos el comparador "<" para que la función "sort" de una lista ordene la lista
     decrecientemente.
inline int operator < (const VoDVirtualServerUsers &usuario) const { return(this->BWout>
     usuario.BWout);};

inline int operator == (const VoDVirtualServerUsers &usuario) const { return(this->
     IdUsuario == usuario.IdUsuario);}; };


typedef list<VoDVirtualServerUsers> TListVoDVirtualServerUsers; typedef list<
     VoDVirtualServerUsers>::iterator TListVoDVirtualServerUsersIterator;
#endif
```

Anexo H - Clase VirtualServerInf

Esta es la encargada de agrupar todos los datos que debe mantener un manager para gestionar su servidor virtual.

```
#ifndef __VirtualServerinf_H__
#define __VirtualServerinf_H__
#include <iostream>
#include <list>
#include <VoDBase.h>
#include <VoDTipos.h>
#include <VoDMetadatos.h>
//#include <VoDServidor.h>

 //Para evitar situaciones de recursividad durante la compilación no podemos realizar un
      include de VoDservidor
```

```
#include <VoDUsuarioActivo.h>
#include <VoDPoliticaServicio.h>
//#include <stl_list.h>
#include <VoDVirtualServerUsers.h> //By DynaPeer

/** Clase que implementa los datos referentes a un ServidorVirtual que guardará un
     manager. By DynaPeer */

class VirtualServerInf{
        int sizeOfVS;
        TListVoDVirtualServerUsers VS;

        public:
                inline VirtualServerInf(){ //Constructora.
                    sizeOfVS=0;
                }
                inline ~VirtualServerInf(){ //Destructora.
                }

                inline void inputUser(char Ip[50], TIdPuerto port, TPeerRol rol, bool
                     supmul, TUserId iduser, TAnchoBandaServicio in, TAnchoBandaServicio
                     out, TBloques buf){

                        VoDVirtualServerUsers usuario; //creamos un nuevo usuario e
                             inicializamos sus campos. // Seguidamente introducimos este
                             usuario en la lista de usuarios.
                        usuario.setsupportMulticast(supmul);
                        usuario.setRol(rol);
                        usuario.setIdUsuario(iduser);
                        usuario.setBWout(out);
                        usuario.setBWin(in);
                        usuario.setIp(Ip);
                        usuario.setPuerto(port);
                        usuario.setBuffer(buf);
                        VS.push_back(usuario);
                        sizeOfVS++;                }

                inline VirtualServerInf algOrdenDescendente(){
                        VS.sort();                 }
                inline void removeUser(TUserId usuario){
                TListVoDVirtualServerUsersIterator Iterador;
                        for ( Iterador = VS.begin(); Iterador != VS.end(); Iterador++ )
                                                                {
                              if ( Iterador->getIdUsuario() == usuario )

                                   {
                                    Iterador = VS.erase(Iterador);
                              }
                        }
                  }

                inline char* getIpUser(TUserId us){ TListVoDVirtualServerUsersIterator
                     Iterador;
                        for ( Iterador = VS.begin(); Iterador != VS.end(); Iterador++ )
                                                                {
                             if ( Iterador->getIdUsuario() == us ) {

                                  return Iterador->getIp();
                             }
                        }
                 }

               inline TPortId getPuertoUser(TUserId us){
                    TListVoDVirtualServerUsersIterator Iterador;
                        for ( Iterador = VS.begin(); Iterador != VS.end(); Iterador++ )
                                                                {
                              if ( Iterador->getIdUsuario() == us )

                                   {
                                    return Iterador->getPuerto();
                              }
                        }
               }

               inline TBloques getBufferUser(TUserId us){
                   TListVoDVirtualServerUsersIterator Iterador;
```

```
                    for ( Iterador = VS.begin(); Iterador != VS.end(); Iterador++ )
                                                                        {
                        if ( Iterador->getIdUsuario() == us )
                        {
                            return Iterador->getBuffer();
                        }
                    }
                }

                inline int getSize(){ return sizeOfVS; }
                inline TListVoDVirtualServerUsers getVS(){ return VS; }

                inline void imprimirListaUsuarios(){
                    TListVoDVirtualServerUsersIterator Iterador;
                    cout<<"[DEBUG][VirtualServerInf] ####### Lista de usuarios
                         presentes en el VS #######"<<endl;
                    for ( Iterador = VS.begin(); Iterador != VS.end(); Iterador++ )
                                                                        {
        cout<<"[DEBUG][VirtualServerInf]"<<"Ip: "<<Iterador->getIp()<<" Puerto: "<<
             Iterador->getPuerto()<<endl;
                    }
        cout<<"[DEBUG][VirtualServerInf] ################## FIN DEL LISTADO
             ##################"<<endl;
                  }

                inline VoDVirtualServerUsers getLastUser(){ return VS.back(); }
                inline VoDVirtualServerUsers getFirstUser(){ return VS.front(); }


};//Fin de la clase VirtualServerInf

typedef class VirtualServerInf TVirtualServerInf, *PtrVirtualServerInf;
typedef list<class VirtualServerInf> TListVirtualServerInf;
typedef list<class VirtualServerInf>::iterator TListVirtualServerInfIterator;

#endif
```

Anexo I - VoDVirtualServer

Define las características de un servidor virtual tales como su manager, que vídeo reproduce etc. El servidor principal implementa una lista de objetos de esta clase para mantener información sobre todos los servidores virtuales presentes en la red.

```
#ifndef __VoDVirtualServer_H__
#define __VoDVirtualServer_H__
#include <iostream>
#include <list>
#include <VoDBase.h>
#include <VoDTipos.h>
#include <VoDMetadatos.h>
#include <VoDUsuarioActivo.h>
#include <VoDPoliticaServicio.h>
#include "VoDServidor.h"

/** Clase que implementa la información basica de un Servidor Virtual y operaciones
     sobre ellos. */ class VoDVirtualServer {

        private:
                bool                            supportMulticast;
                TVideoId                        video;
              TDescriptorUsuarioActivo          manager;        //nos sirve para saber en
                   cualquier momento quien es el manager de un servidor virtual.
              list<TDescriptorUsuarioActivo>    users;  //mantiene una lista con los
                   integrantes del servidor virtual
                TIdBloque                       inicio, fin;
```

```
        public:
             VoDVirtualServer() {};
         inline VoDVirtualServer(bool supportsMulticast, TVideoId vid,
              TDescriptorUsuarioActivo manag, TDescriptorUsuarioActivo usuario, TIdBloque
               ini, TIdBloque final){
              supportMulticast=supportsMulticast;
                  video=vid;
                  manager=manag;
                  users.push_back ( usuario );
                  inicio=ini;
                  fin=final;

                  cout<<"[DEBUG][VoDVirtualServer] Nuevo Servidor Virtual Creado"<<endl;
                                                        cout<<"[DEBUG][VoDVirtualServer]
                       Manager del VS:"<<manager<<endl;
                                                              cout<<"[DEBUG][VoDVirtualServer
                       ] Video del VS:"<<vid<<endl;
                                                                  cout<<"[DEBUG][
                       VoDVirtualServer] OffsetInicio: "<<ini<<" OffsetFinal: "<<fin<<endl
                       ;
             }

   ~VoDVirtualServer() {};

inline void setsupportMulticast ( bool supportsMulticast ){ supportMulticast=
     supportsMulticast; }
inline void setVideo ( TVideoId vid ){ video=vid; }
inline void setManager ( TDescriptorUsuarioActivo manag ) { manager=manag; }
inline void inputUser ( TDescriptorUsuarioActivo usuario ) {
       users.push_back ( usuario );
       cout<<"[DEBUG] [VoDVirtualServer] Nuevo usuario introducido."<<endl; }
inline void setInicio ( TIdBloque ini ) { inicio=ini; }
inline void setFin ( TIdBloque final ) { fin=final; }
inline bool getsupportMulticast() { return supportMulticast; }
inline TVideoId getVideoId() { return video; }
inline TDescriptorUsuarioActivo getManager() { return manager; }
inline list<TDescriptorUsuarioActivo> getListaUsuarios() { return users; }
inline TDescriptorUsuarioActivo ObtenerUsuario(){ return *(users.begin()); }
inline TIdBloque getInicio() { return inicio; }
inline TIdBloque getFin() { return fin; }
inline TListDescriptorUsuarioActivoIterator searchUser(TDescriptorUsuarioActivo& user){
         TListDescriptorUsuarioActivoIterator iter;
         for(iter=users.begin(); iter!=users.end(); iter++){
            if (*iter==user){
                  return iter;
            }
          }
    return users.end();
}

inline bool UsuarioEncontrado(TDescriptorUsuarioActivo& user){
   TListDescriptorUsuarioActivoIterator iter;
         for(iter=users.begin(); iter!=users.end(); iter++){
            if (*iter==user){
                  return true;
            }
         }
    return false;
}

inline void removeUser (TDescriptorUsuarioActivo& user){
         users.remove(user);
         cout<<"[DEBUG] [VoDVirtualServer] Usuario eliminado"<<endl;
}

inline TListDescriptorUsuarioActivoIterator getIterUsers(){
         TListDescriptorUsuarioActivoIterator iter;
         iter=users.begin();
         return iter;
}

inline TDescriptorUsuarioActivo getUltimoUser(){ return users.back(); }

inline long int getInicioVentana(){
TIdBloque inicioventana=0;
```

```
 if (users.back().GetPtrUsuario()->GetNumeroCanalesUsuario()>0)            inicioventana
      =((users.back().GetPtrUsuario()->GetCanalUsuario(1).GetPtrCanal()->
      GetUltimoBloqueEnviado())*DBytesBloque)-((users.back().GetPtrUsuario())->
      GetBufferClienteP2P());

//(users.back().GetPtrUsuario()->GetUltimoBloqueReproducido())-((users.back().
     GetPtrUsuario())->GetVentanaColaboracionP2P());


 cout<<"[DEBUG] [VoDVirtualServer] Inicio ventana colaboracion: "<<inicioventana<<endl;
 return inicioventana;
}

inline TIdBloque getFinVentana(){
     return (users.back().GetPtrUsuario())->GetUltimoBloqueReproducido();
}

inline void imprimirInfoVS(){

 TIdBloque inicioventana=0;
 cout<<"[DEBUG][VoDVirtualServer] Manager del VS:"<<manager<<endl;
                                                 cout<<"[DEBUG][VoDVirtualServer] Video del VS:"
     <<video<<endl;
 cout<<"[DEBUG][VoDVirtualServer] OffsetInicio: "<<inicio<<" OffsetFinal: "<<fin<<endl;
  if (users.back().GetPtrUsuario()->GetNumeroCanalesUsuario()>0)
                                                 inicioventana=((users.back().GetPtrUsuario()->
       GetCanalUsuario(1).GetPtrCanal()->GetUltimoBloqueEnviado())*DBytesBloque)-((users.
       back().GetPtrUsuario())->GetBufferClienteP2P());

       cout<<"[DEBUG] [VoDVirtualServer] Inicio ventana colaboracion: "<<inicioventana<<
            endl<<endl;
}

inline int operator < (const VoDVirtualServer &A) const { return(this->video<A.video);};
};


typedef list<class VoDVirtualServer> TListVoDVirtualServer;
typedef list<class VoDVirtualServer>::iterator TListVoDVirtualServerIterator;

#endif //fin de la clase VirtualServer
```

Anexo J - Mensajes añadidos

A continuación se muestran las definiciones de los nuevos mensajes añadidos.

```
class VoDMensajePeticionesParciales: public VoDMensaje {
  TUserId user; //Usamos directamente este tipo y no el descriptor de usuario para
       evitarnos una conversio, aunque podríamos usar de la misma manera el descriptor y
       despues realizar un "GetPtrUsuario"
   TVideoId         video;
   TVideoOffset     ini, fin;
   TPosicionTurno   ranTur;
   TIdBloque        BRini, BRfin;
   TIdBloque        MRango;
   bool             mand;

        public:
                VoDMensajePeticionesParciales(){ Mensaje = MPeticionesParciales;
                     Iniciado=False; };          //Iniciado igual a FALSE nos indica que
                     no inicia el servidor.

                ~VoDMensajePeticionesParciales(){};

                //Sets
                inline void SetIdUser(TUserId manager){this->user=manager;}
                inline void SetIdVideo(TVideoId vid){this->video=vid;}
                inline void SetOffsetIni(TVideoOffset inicio){this->ini=inicio;}
                inline void SetOffsetFin(TVideoOffset final){this->fin=final;}
```

```
                inline void SetRanuraTurno(TPosicionTurno turno){this->ranTur=turno;}
                inline void SetBRini(TIdBloque brangoini){this->BRini=brangoini;}
                inline void SetBRfin(TIdBloque brangofin){this->BRfin=brangofin;}
                inline void SetModRango(TIdBloque modrang){this->MRango=modrang;}
                                                        inline void SetMandatory(bool
                    mandatory){this->mand=mandatory;}

                //Gets
                inline TVideoId getIdVideo(){return video;}
                inline TVideoOffset getOffsetIni(){return ini;}
                inline TVideoOffset getOffsetFin(){return fin;}
                inline TPosicionTurno getPosicionTurno(){return ranTur; }
                inline TIdBloque getBRangoIni(){return BRini;}
                inline TIdBloque getBRangoFin(){return BRfin;}
                inline TIdBloque getModRango(){return MRango;}
                inline bool getMandatory(){return mand; }
                inline TUserId getUsuario(){return user;}

};
typedef VoDMensajePeticionesParciales TVoDMensajePeticionesParciales, *
    PtrVoDMensajePeticionesParciales;


#
    ##############################################################################################


class VoDMensajeSetMandatory: public VoDMensaje{
        private:
                bool mand;
        public:
                VoDMensajeSetMandatory(){ Mensaje=MSetMandatory; Iniciado=True; };
                                                        ~VoDMensajeSetMandatory(){};

                inline void setValue(bool value){ this->mand=value; }
                inline bool getValue(){ return mand; }

}; //Fin de la clase VoDMensajeSetMandatory
typedef VoDMensajeSetMandatory TVoDMensajeSetMandatory, *PtrVoDMensajeSetMandatory;


#
    ##############################################################################################


class VoDMensajeDireccionVS: public VoDMensaje {
    PtrUsuarioActivo        IdManager; //Usamos directamente este tipo y no el
        descriptor de usuario para evitarnos una conversio, aunque podríamos usar de la
        misma manera el descriptor y despues realizar un "GetPtrUsuario"
    TVideoId        video;
    TVideoOffset    ini, fin;
    TPosicionTurno  ranTur;
    TIdBloque       BRini, BRfin;
    TIdBloque       MRango;
    bool            mand;
    TPortId         PuertoEntrada;
    char            Ip[50];
    TBloques        buffer;
        public:
                VoDMensajeDireccionVS(){ Mensaje = MDireccionVS; Iniciado=False; }; //
                    Iniciado igual a FALSE nos indica que no inicia el servidor.
                 ~VoDMensajeDireccionVS(){};

                //Sets
                inline void SetIdManager(PtrUsuarioActivo manager){this->IdManager=
                    manager;}                           inline void SetIdVideo(TVideoId
                    vid){this->video=vid;}
                inline void SetOffsetIni(TVideoOffset inicio){this->ini=inicio;}
                inline void SetOffsetFin(TVideoOffset final){this->fin=final;}
                inline void SetRanuraTurno(TPosicionTurno turno){this->ranTur=turno;}
                inline void SetBRini(TIdBloque brangoini){this->BRini=brangoini;}
                inline void SetBRfin(TIdBloque brangofin){this->BRfin=brangofin;}
                inline void SetModRango(TIdBloque modrang){this->MRango=modrang;}
                                                        inline void SetMandatory(bool
                    mandatory){this->mand=mandatory;}
```

```
                inline void SetPuertoEntrada(TPortId puerto){this->PuertoEntrada=puerto
                    ;};
                inline void SetDireccionIp(char direccionIp[]){strcpy(Ip,direccionIp);};
                                                    inline void SetBuffer(TBloques buf){
                    this->buffer=buf;};

                //Gets
                inline TVideoId getIdVideo(){return video;}
                inline TVideoOffset getOffsetIni(){return ini;}
                inline TVideoOffset getOffsetFin(){return fin;}
                inline TPosicionTurno getPosicionTurno(){return ranTur; }
                inline TIdBloque getBRangoIni(){return BRini;}
                inline TIdBloque getBRangoFin(){return BRfin;}
                inline TIdBloque getModRango(){return MRango;}
                inline bool getMandatory(){return mand; }
                inline PtrUsuarioActivo GetIdManager(){return IdManager;}
                inline TPortId getPuertoEntrada(){return PuertoEntrada;};
                inline char* getDireccionIp(){return Ip;};
                inline TBloques getBuffer(){return buffer;};

};//Fin de la clase VoDMensajeDireccionVS
typedef VoDMensajeDireccionVS TVoDMensajeDireccionVS, *PtrVoDMensajeDireccionVS;


#
    ##################################################################################################


class VoDMensajeAsignarRol: public VoDMensaje {
      TPeerRol          Rol;
        public:
           VoDMensajeAsignarRol(){ Mensaje = MAsignarRol; Iniciado=False; }; //Iniciado
                igual a FALSE nos indica que el msg lo inicia el servidor.
           ~VoDMensajeAsignarRol(){};

                //Sets & Gets
                inline void SetRol(TPeerRol rol){ Rol=rol; }
                inline TPeerRol GetRol(){ return Rol; }

};//Fin de la clase VoDMensajeAsignarRol
typedef VoDMensajeAsignarRol TVoDMensajeAsignarRol, *PtrVoDMensajeAsignarRol;


#
    ##################################################################################################


class VoDMensajePeticionesDireccionIp: public VoDMensaje {
    TUserId user; //Usamos directamente este tipo y no el descriptor de usuario para
         evitarnos una conversio, aunque podríamos usar de la misma manera el descriptor
         y despues realizar un "GetPtrUsuario"
     TVideoId          video;
     TVideoOffset      ini, fin;
     TPosicionTurno    ranTur;
     TIdBloque         BRini, BRfin;
     TIdBloque         MRango;
     bool              mand;

        public:
                VoDMensajePeticionesDireccionIp(){ Mensaje = MPeticioneDireccionIp;
                    Iniciado=False; }; //Iniciado igual a FALSE nos indica que no inicia
                     el servidor.                                                    ~
                    VoDMensajePeticionesDireccionIp(){};

                //Sets
                inline void SetIdUser(TUserId manager){this->user=manager;}
                inline void SetIdVideo(TVideoId vid){this->video=vid;}
                inline void SetOffsetIni(TVideoOffset inicio){this->ini=inicio;}
                inline void SetOffsetFin(TVideoOffset final){this->fin=final;}
                inline void SetRanuraTurno(TPosicionTurno turno){this->ranTur=turno;}
                inline void SetBRini(TIdBloque brangoini){this->BRini=brangoini;}
                inline void SetBRfin(TIdBloque brangofin){this->BRfin=brangofin;}
                inline void SetModRango(TIdBloque modrang){this->MRango=modrang;}
                                                    inline void SetMandatory(bool
                    mandatory){this->mand=mandatory;}
```

```
                //Gets
                inline TVideoId getIdVideo(){return video;}
                inline TVideoOffset getOffsetIni(){return ini;}
                inline TVideoOffset getOffsetFin(){return fin;}
                inline TPosicionTurno getPosicionTurno(){return ranTur; }
                inline TIdBloque getBRangoIni(){return BRini;}
                inline TIdBloque getBRangoFin(){return BRfin;}
                inline TIdBloque getModRango(){return MRango;}
                inline bool getMandatory(){return mand; }
                inline TUserId getUsuario(){return user;}

};
typedef VoDMensajePeticionesDireccionIp TVoDMensajePeticionesDireccionIp, *
    PtrVoDMensajePeticionesDireccionIp;


#
    ##############################################################################################


class VoDMensajeRedireccionPlay: public VoDMensaje {
 TUserId user; //Usamos directamente este tipo y no el descriptor de usuario para
     evitarnos una conversio, aunque podríamos usar de la misma manera el descriptor y
     despues realizar un "GetPtrUsuario"
    TVideoId        video;
    TVideoOffset    ini, fin;
    TPosicionTurno  ranTur;
    TIdBloque       BRini, BRfin;
    TIdBloque       MRango;
    bool            mand;
    TPortId         PuertoEntrada;
    char            Ip[50];
    TBloques        buffer;

        public:
                VoDMensajeRedireccionPlay(){ Mensaje = MRedireccionPlay; Iniciado=False;
                    }; //Iniciado igual a FALSE nos indica que no inicia el servidor.
                                                                                    ~
                    VoDMensajeRedireccionPlay(){};

                //Sets
                inline void SetIdManager(TUserId manager){this->user=manager;}
                inline void SetIdVideo(TVideoId vid){this->video=vid;}
                inline void SetOffsetIni(TVideoOffset inicio){this->ini=inicio;}
                inline void SetOffsetFin(TVideoOffset final){this->fin=final;}
                inline void SetRanuraTurno(TPosicionTurno turno){this->ranTur=turno;}
                inline void SetBRini(TIdBloque brangoini){this->BRini=brangoini;}
                inline void SetBRfin(TIdBloque brangofin){this->BRfin=brangofin;}
                inline void SetModRango(TIdBloque modrang){this->MRango=modrang;}
                                                        inline void SetMandatory(bool
                     mandatory){this->mand=mandatory;}
                inline void SetPuertoEntrada(TPortId puerto){this->PuertoEntrada=puerto
                    ;};
                inline void SetDireccionIp(char direccionIp[]){strcpy(Ip,direccionIp);};
                                                    inline void SetBuffer(TBloques buf){
                    this->buffer=buf;};

                //Gets
                inline TVideoId getIdVideo(){return video;}
                inline TVideoOffset getOffsetIni(){return ini;}
                inline TVideoOffset getOffsetFin(){return fin;}
                inline TPosicionTurno getPosicionTurno(){return ranTur; }
                inline TIdBloque getBRangoIni(){return BRini;}
                inline TIdBloque getBRangoFin(){return BRfin;}
                inline TIdBloque getModRango(){return MRango;}
                inline bool getMandatory(){return mand; }
                inline TUserId getUsuario(){return user;}
                inline TPortId getPuertoEntrada(){return PuertoEntrada;};
                inline char* getDireccionIp(){return Ip;};
                inline TBloques getBuffer(){return buffer;};

};
typedef VoDMensajeRedireccionPlay TVoDMensajeRedireccionPlay, *
    PtrVoDMensajeRedireccionPlay;
```

```
#
    ##########################################################################################################

class VoDMensajeNuevoPeerVS: public VoDMensaje {
        bool                    supportMulticast;
        TPeerRol                Rol;
        TUserId                 IdUsuario;
        TAnchoBandaServicio     BWin;
        TAnchoBandaServicio     BWout;
        char                    ip[50];
        TIdPuerto               puerto;
        TBloques                buffer;

        public:

        VoDMensajeNuevoPeerVS(){ Mensaje = MNuevoPeerVS; Iniciado=False; };
                                            ~VoDMensajeNuevoPeerVS(){};

        //Sets & Gets
        inline void setsupportMulticast(bool supmul){supportMulticast=supmul;}
        inline void setRol(TPeerRol rol){Rol=rol;}
        inline void setIdUsuario(TUserId usuario){IdUsuario=usuario;}
        inline void setBWin(TAnchoBandaServicio bwin){BWin=bwin;}
        inline void setBWout(TAnchoBandaServicio bwout){BWout=bwout;}
        inline void setIp(char Ip[50]){strcpy(ip, Ip);}
        inline void setPuerto(TIdPuerto port){puerto=port;}
        inline void setBuffer(TBloques buf){buffer=buf;}

        inline bool getsupportMulticast(){return supportMulticast;}
        inline TPeerRol getRol(){return Rol;}
        inline TUserId getIdUsuario(){return IdUsuario;}
        inline TAnchoBandaServicio getBWin(){return BWin;}
        inline  TAnchoBandaServicio getBWout(){return BWout;}
        inline char* getIp(){return ip;}
        inline TIdPuerto getPuerto(){return puerto;}
        inline TBloques getBuffer(){return buffer;}
};
typedef VoDMensajeNuevoPeerVS TVoDMensajeNuevoPeerVS, *PtrVoDMensajeNuevoPeerVS;
```

Anexo K - Envío de los nuevos mensajes

A continuación se detallan las funciones utilizadas para el envío de los mensajes.

```
TCodigoError VoDThreadControl::EnviarMensajeDireccionVS(PtrUsuarioActivo manager,
     TVideoId vid, TVideoOffset ini, TVideoOffset fin, TPosicionTurno rantur, TIdBloque
     brini, TIdBloque brfin, TIdBloque mrango, bool mand, TPortId puerto, char ip[],
     TBloques buffer ) {

    if (VoDServidor::GetDepuracionActivada()){
            cout << "[Debug_Th_Control][IdU" << GetIdUsuarioActivo() << "][" << IdThread
                 << "]" << "Enviando_direccion_virtual_server_al_cliente" << endl;
                        cout.flush();
    } //end if

        VoDMensajeDireccionVS    vmdvs;
        vmdvs.SetIdManager(manager);
        vmdvs.SetIdVideo(vid);
        vmdvs.SetOffsetIni(ini);
        vmdvs.SetOffsetFin(fin);
        vmdvs.SetRanuraTurno(rantur);
        vmdvs.SetBRini(brini);
        vmdvs.SetBRfin(brfin);
        vmdvs.SetModRango(mrango);
        vmdvs.SetMandatory(mand);
        vmdvs.SetPuertoEntrada(puerto);
        vmdvs.SetDireccionIp(ip);
```

```
        vmdvs.SetBuffer(buffer);

cout<< "#!!_"<<vmdvs.getDireccionIp()<<"puerto:"<<vmdvs.getPuertoEntrada()<< endl;
        if (vmdvs.Enviar(GetIdConexion(), sizeof(vmdvs)) != Ok) return Warning(
            CErrEnviarDireccionVS );
        return Ok;

}//end EnviarMensajeDireccionVSmanagemanager //end By DynaPeer


#
    ##########################################################################################################


//By DynaPeer
TCodigoError VoDThreadControl::EnviarMensajeAsignarRol(TPeerRol  rol){
  if (VoDServidor::GetDepuracionActivada()){
          cout << "[Debug_Th_Control][IdU" << GetIdUsuarioActivo() << "][" << IdThread
              << "]" << "Enviando_nuevo_rol_al_cliente" << endl;
           cout.flush();
  }//end if

        VoDMensajeAsignarRol vmar;
        vmar.SetRol(rol);
        if(vmar.Enviar(GetIdConexion(), sizeof(vmar))!=Ok)return Warning(
            CErrEnviarMensajeAsignarRol );
        return Ok;
}


#
    ##########################################################################################################


TCodigoError VoDThreadControl::EnviarMensajePeticionesParciales(TUserId user, TVideoId
     video, TVideoOffset inicio, TVideoOffset fin, TPosicionTurno ranuraTurno, TIdBloque
     RangoModuloInicio, TIdBloque RangoModuloFinal, TIdBloque ModuloRango){

if (VoDServidor::GetDepuracionActivada()){
          cout << "[Debug_Th_Control][IdU" << GetIdUsuarioActivo() << "][" << IdThread
              << "]" << "Enviando_mensaje_de_peticion_parcial_al_cliente" << endl;
           cout.flush();
}//end if

        VoDMensajePeticionesParciales vmpp;
        vmpp.SetIdUser(user);
        vmpp.SetIdVideo(video);
        vmpp.SetOffsetIni(inicio);
        vmpp.SetOffsetFin(fin);
        vmpp.SetRanuraTurno(ranuraTurno);
        vmpp.SetBRini(RangoModuloInicio);
        vmpp.SetBRfin(RangoModuloFinal);
        vmpp.SetModRango(ModuloRango);
        if(vmpp.Enviar(GetIdConexion(),sizeof(vmpp))!=Ok)return Warning(
            CErrEnviarMensajePeticionParcial );
        return Ok;
}


#
    ##########################################################################################################


TCodigoError VoDThreadControl::EnviarPeticionIp(TUserId user, TVideoId video,
     TVideoOffset ini, TVideoOffset fin, TPosicionTurno posturn, TIdBloque bri, TIdBloque
      brf, TIdBloque mr, bool man){

        VoDMensajePeticionesDireccionIp vmpdi;
        vmpdi.SetIdUser(user);
        vmpdi.SetIdVideo(video);
        vmpdi.SetOffsetIni(ini);
        vmpdi.SetOffsetFin(fin);
        vmpdi.SetRanuraTurno(posturn);
        vmpdi.SetBRini(bri);
```

```
        vmpdi.SetBRfin(brf);
        vmpdi.SetModRango(mr);
        vmpdi.SetMandatory(man);

        if (vmpdi.Enviar(GetIdConexion(), sizeof(vmpdi)) != Ok) return Warning(
            CErrEnviarPeticionIp );
        return Ok; }

#
    ##########################################################################################################


TCodigoError VoDThreadControl::EnviarMensajeRedireccionPlay(TUserId user, TVideoId vid,
    TVideoOffset ini, TVideoOffset fin, TPosicionTurno rantur, TIdBloque brini,
    TIdBloque brfin, TIdBloque mrango, bool mand, TPortId puerto, char ip[], TBloques
    buffer ) {

        if (VoDServidor::GetDepuracionActivada())        {
            cout << "[Debug_Th_Control][IdU" << GetIdUsuarioActivo() << "][" << IdThread
                 << "]" << "Enviando_direccion_virtual_server_al_cliente" << endl;
             cout.flush();
}//end if

        VoDMensajeRedireccionPlay        vmrp;
        vmrp.SetIdManager(user);
        vmrp.SetIdVideo(vid);
        vmrp.SetOffsetIni(ini);
        vmrp.SetOffsetFin(fin);
        vmrp.SetRanuraTurno(rantur);
        vmrp.SetBRini(brini);
        vmrp.SetBRfin(brfin);
        vmrp.SetModRango(mrango);
        vmrp.SetMandatory(mand);
        vmrp.SetPuertoEntrada(puerto);
        vmrp.SetDireccionIp(ip);
        vmrp.SetBuffer(buffer);

        if (vmrp.Enviar(GetIdConexion(), sizeof(vmrp)) !=Ok)return Warning(
            CErrEnviarRedireccionPlay );
        return Ok;
}//end EnviarMensajeRedireccionPlay

#
    ##########################################################################################################


TCodigoError VoDThreadControl::EnviarMensajeNuevoPeerVS(char Ip[50], TIdPuerto port,
    TPeerRol rol, bool supmul, TUserId iduser, TAnchoBandaServicio in,
    TAnchoBandaServicio out, TBloques buf) {

        VoDMensajeNuevoPeerVS vmnpv;
        vmnpv.setIp(Ip);
        vmnpv.setPuerto(port);
        vmnpv.setRol(rol);
        vmnpv.setsupportMulticast(supmul);
        vmnpv.setIdUsuario(iduser);
        vmnpv.setBWin(in);
        vmnpv.setBWout(out);
        vmnpv.setBuffer(buf);

if (vmnpv.Enviar(GetIdConexion(), sizeof(vmnpv)) != Ok)return Warning(
    CErrEnviarNuevoPeerVS);
        return Ok;
}
```